

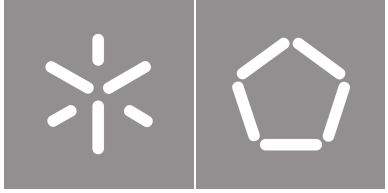
Universidade do Minho
Escola de Engenharia

Bruno Vilaça de Sá

CAMES: Capability Architectures for Mixed-Criticality Embedded Systems

Maio de 2026

CAMES: Capability Architectures for Mixed-Criticality Embedded Systems
Bruno Sá
UMinho | 2026



Universidade do Minho
Escola de Engenharia

Bruno Vilaça de Sá

CAMES: Capability Architectures for Mixed-Criticality Embedded Systems

Tese de Doutoramento
Programa Doutoral em Engenharia Eletrónica e Computadores

Trabalho efetuado sob a orientação de:
Investigador Principal Doutor Sandro Emanuel Salgado Pinto
Professor Doutor João Luís Marques Pereira Monteiro

Maio de 2026

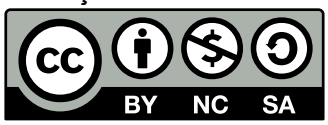
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

CC BY-NC-SA 4.0

`https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en`

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.2) .

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Sandro Pinto, for his unwavering support, guidance, and mentorship throughout this journey. Your dedication, resilience, and relentless commitment to research have been a constant source of inspiration to me. Thank you for the countless hours of support, encouragement, and technical discussions that shaped both this work and my growth as a researcher.

I would also like to thank all my colleagues at ESRG for their support, friendship, and camaraderie throughout these years. Whether through collaboration, technical discussions, shared travels, or simply moments spent together in the lab, each of you contributed to making this journey both rewarding and memorable.

A special thank you goes to my friends Diogo and Sofia. Our weekly quiz nights became much more than a simple tradition; they were moments of support, laughter, and balance during the most demanding periods of this PhD. Thank you for always listening, encouraging me, and being there through every stage of this journey.

To my parents, my brother, my parents-in-law, and my grandmother Joana, I am profoundly grateful for your unconditional love, encouragement, and belief in me. Although my grandmother is no longer with us, her strength, values, and memory will always remain an inspiration to me. Your support gave me the strength to keep moving forward during the most challenging moments. This achievement would not have been possible without all of you.

Finally, and most importantly, I want to thank my wife, Catarina. For the past thirteen years, you have been my rock, my safe place, and my greatest source of strength. Thank you for your endless patience, unconditional love, encouragement, and understanding throughout every step of this journey. Your support carried me through the long days, sleepless nights, and difficult moments. I am deeply grateful to share my life with you, and this achievement is as much yours as it is mine.

To everyone who has been part of this journey, thank you.

This work was supported by national funds through the FCT doctoral grant 2021.07707.BD.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

DECLARATION ON THE USE OF ARTIFICIAL INTELLIGENCE (AI) TOOLS

I hereby declare having acted this academic work with integrity and confirm that I have not resorted to any form of undue use of AI tools, nor to the falsification or omission of information, sources, data, results or processes in any of the stages leading to its elaboration.

I further declare that any use of AI tools was done in a transparent and responsible manner, being duly noted in the work, and that I critically verified the accuracy of the content produced with their support, assuming full responsibility for what I submit.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho, as well as the applicable standards of academic integrity and data protection.

_____, _____

(Place) (Date)

_____

(Bruno Vilaça de Sá)

Resumo

CAMES: Arquiteturas Baseadas em Capacidades para Sistemas Embebidos de Criticidade Mista

Os sistemas embebidos e de criticidade elevada têm vindo a tornar-se mais complexos devido às crescentes exigências de integração e conectividade. Em simultâneo, as pressões de mercado para reduzir dimensão, peso, consumo energético e custo (SWaP-C) têm impulsionado a consolidação de aplicações de criticidade mista numa única plataforma. Neste contexto, a virtualização tem-se afirmado como uma tecnologia de referência para a consolidação de sistemas de criticidade mista, oferecendo isolamento espacial e temporal e facilitando o cumprimento de normas de segurança e fiabilidade, como a ISO 26262. Embora os hipervisores tradicionais forneçam isolamento eficaz entre máquinas virtuais (VMs), garantir proteção com mínima granularidade dentro de uma VM ou no próprio hipervisor requer mecanismos adicionais, frequentemente complexos e dispendiosos. Esta limitação torna-se relevante em sistemas desenvolvidos em linguagens sem garantias de segurança de memória, como C e C++, onde vulnerabilidades de memória podem comprometer as garantias de isolamento. A arquitetura Capability Hardware Enhanced RISC Instructions (CHERI) mitiga estas limitações através da introdução de mecanismos de proteção de memória de granularidade fina impostos por hardware e de compartimentação granular através de `capabilities`. Ao contrário de linguagens seguras, como Rust, CHERI permite uma adoção incremental em sistemas baseados em C/C++, sem exigir a reescrita integral do código existente, embora implique alterações na pilha de software (por exemplo, no toolchain) e suporte adicional de hardware.

Esta tese estabelece uma base para segurança baseada em `capabilities` e virtualização em sistemas embebidos RISC-V de criticidade mista através de três contributos. Em primeiro lugar, realizámos a primeira análise sistemática do ecossistema CHERI e identificámos lacunas em setores críticos, como o setor automóvel. Em segundo lugar, estendemos o processador RISC-V CVA6 com suporte para virtualização e explorámos melhorias microarquiteturais destinadas a aumentar a eficiência da virtualização através de exploração do espaço de desenho. Por fim, desenvolvemos a primeira pilha completa open-source que integra CHERI com virtualização em RISC-V. Em particular, implementámos CHERI numa plataforma baseada no CVA6 com suporte para virtualização e estendemos o hipervisor Bao para suportar CHERI em RISC-V, tornando-o o primeiro hipervisor com suporte para CHERI nesta arquitetura, avaliando o esforço de integração necessário e os benefícios de segurança obtidos.

Palavras-chave: CHERI, CVA6, RISC-V, Sistemas de Criticidade Mista, Virtualização

Abstract

CAMES: Capability Architectures for Mixed-Criticality Embedded Systems

Embedded and safety-critical systems are growing in complexity due to increasing demands for integration and connectivity. At the same time, market pressures to minimize Size, Weight, Power and Cost (SWaP-C) are pushing towards the consolidation of mixed-criticality workloads onto a single platform (e.g., the co-execution of real-time and general-purpose operating systems (OSs)). In this context, virtualization has emerged as a de facto technology for consolidating mixed-criticality workloads, providing spatial and temporal isolation to support compliance with safety and security standards (e.g., ISO 26262). Although traditional hypervisors effectively isolate Virtual Machines (VMs) from each other, enforcing fine-grained protection within a VM or inside the hypervisor itself requires additional mechanisms, which are often complex and costly. This limitation becomes particularly relevant in systems written in memory-unsafe languages, such as C and C++, where memory-safety vulnerabilities may compromise isolation guarantees. Capability Hardware Enhanced RISC Instructions (CHERI) mitigates these issues with fine-grained hardware-enforced memory protection and lightweight object-level compartmentalization via architectural capabilities. Unlike memory-safe languages such as Rust, CHERI supports incremental adoption without a full rewrite of C/C++ codebases, making it suitable for legacy systems, though it requires intrusive changes to the software stack (e.g., toolchain) and hardware support.

This thesis establishes a baseline for capability-based security and virtualization in mixed-criticality RISC-V embedded systems through three contributions. First, we provided the first comprehensive survey of the CHERI ecosystem, classifying hardware, toolchains, OSs, and runtimes, and identifying key gaps for safety-critical domains such as automotive. Second, we extended the CVA6 RISC-V core with hypervisor support and explored microarchitectural enhancements to improve virtualization efficiency through design space exploration. Finally, we delivered the first open-source end-to-end system stack integrating CHERI with RISC-V virtualization. Specifically, we implemented CHERI on a CVA6-based platform with virtualization support and extended the open-source Bao hypervisor to be CHERI-aware, the first hypervisor to support it on RISC-V, and evaluated the implementation effort and the resulting security benefits.

Keywords: CHERI, CVA6, Mixed-Criticality Systems, RISC-V, Virtualization

Contents

List of Figures **xiii**

List of Tables **xv**

Acronyms **xvii**

I Dissertation Summary **1**

1 Introduction **2**

1.1 Motivation 2

1.1.1 The Evolution of Embedded Systems 2

1.1.2 Memory Safety for Embedded Systems 3

1.1.3 Isolation Technologies for Embedded Systems 5

1.1.4 Hardware Capability-based Protection: Security-by-Design 6

1.1.5 CHERI for Next-Generation Secure Systems 7

1.2 Contributions 9

1.2.1 Research Challenges 9

1.2.2 Research Questions 10

1.2.3 Main Contributions 11

1.2.4 Other Contributions 14

1.3 Thesis Outline 14

2 Background **15**

2.1 Virtualization 15

2.1.1 Virtualization Techniques 16

2.1.2 Hypervisor Taxonomy 16

2.1.3 Virtualization in Mixed-Criticality Systems 17

2.2 Capability Systems . . . 20
2.2.1 What is a Capability? . . . 20
2.2.2 Capabilities vs. Access Control Lists . . . 21
2.2.3 Evolution of Capability Systems . . . 21
2.3 CHERI . . . 23
2.3.1 CHERI Protection Model . . . 23
2.3.2 CHERI and Legacy Protection Mechanisms . . . 24
2.3.3 Architectural Support: RISC-V and Arm Morello . . . 25
2.4 RISC-V . . . 25
2.4.1 RISC-V Market Growth . . . 26
2.4.2 RISC-V ISA Overview . . . 26
2.4.3 CVA6 . . . 29

3 State of the Art **32**
3.1 Memory Safety: Relevance, Exploits, and Mitigation Techniques . . . 32
3.1.1 Taxonomy of Memory Safety Violations . . . 33
3.1.2 Taxonomy of Memory Safety Protection Techniques . . . 34
3.1.3 Memory Safety Enforcement and Mitigation Techniques . . . 34
3.1.4 Memory Safety Hardening Techniques . . . 42
3.2 Compartmentalization and Isolation Technologies . . . 46
3.2.1 Microkernels and Component-based Operating Systems . . . 46
3.2.2 Virtualization for Mixed-Criticality Systems . . . 48
3.2.3 Microarchitectural and Memory-System Isolation . . . 50
3.2.4 Trusted Execution Environments and Confidential Computing . . . 52
3.3 Discussion and Open Gaps . . . 55
3.3.1 CHERI in the Memory-Safety Landscape . . . 55
3.3.2 CHERI vs. Other Compartmentalization and Isolation Technologies . . . 59

4 Concluding Remarks **63**
4.1 Conclusions . . . 63
4.2 Future Work . . . 66
4.2.1 Memory Subsystem Improvements . . . 67
4.2.2 Architectural and Microarchitectural Directions for CHERI . . . 67
4.2.3 Virtualization, I/O Isolation, and Compartmentalization . . . 69
4.2.4 Performance, Multicore Evaluation, and PPA Analysis . . . 69

II Publications **70**

5 Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology **72**

5.1 Introduction . . . 73
5.2 Capability Systems: a Historical glance . . . 76
5.2.1 Hardware Capability Systems . . . 77
5.2.2 Software Capabilities . . . 81
5.3 CHERI Protection Model . . . 83
5.3.1 Capability Composition, Core Properties, and Misuse Handling . . . 83
5.3.2 Hybridization . . . 84
5.3.3 Sealing Mechanism . . . 85
5.4 CHERI Hardware . . . 86
5.4.1 CHERI-Aware ISAs . . . 86
5.4.2 Capability Compression . . . 87
5.4.3 Capability Tagging . . . 88
5.4.4 CHERI CPUs, SoC, Platforms . . . 88
5.5 CHERI Software . . . 92
5.5.1 Compilers and Linkers . . . 93
5.5.2 Programming Languages and Tools . . . 93
5.5.3 OSs, RTOSs and Microkernels . . . 95
5.5.4 CHERI Runtime System . . . 98
5.5.5 Libraries and Frameworks . . . 101
5.5.6 Applications . . . 102
5.6 CHERI Application Domains & Use-cases . . . 103
5.6.1 CHERI for Edge-to-Cloud systems . . . 103
5.6.2 CHERI for Mixed Criticality Systems . . . 104
5.7 CHERI Future Directions . . . 105
5.7.1 CHERI for Heterogeneous Computing . . . 105
5.7.2 CHERI for Large-Scale Applications . . . 106
5.7.3 CHERI for I/O . . . 106
5.7.4 CHERI for Hypervisors . . . 107
5.7.5 CHERI for TEE . . . 107
5.7.6 CHERI for Side-Channel Attacks . . . 108
5.8 Conclusion . . . 109

6 CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration **110**

6.1 Introduction . . . 111
6.2 Background . . . 113
6.2.1 Virtualization Technology . . . 114
6.2.2 RISC-V Privileged Specification . . . 114
6.2.3 RISC-V Virtualization . . . 114
6.2.4 Nested Memory Management Unit (Nested-MMU) . . . 115
6.2.5 RISC-V “stimecmp/vstimecmp” Extension (Sstc) . . . 116
6.2.6 CVA6 . . . 116
6.3 CVA6 “From Scratch” Virtualization Support . . . 116
6.3.1 Hypervisor and Virtual Supervisor Execution Modes . . . 117
6.3.2 Hypervisor Load/Store Instructions . . . 118
6.3.3 Nested Page-Table Walker . . . 119
6.3.4 Virtualization-aware TLBs (vTLB) . . . 119
6.3.5 Microarchitectural extension #1 - GTLB . . . 120
6.3.6 Microarchitectural extension #2 - L2 TLB . . . 121
6.3.7 Sstc Extension . . . 122
6.4 Design Space Exploration: Evaluation . . . 122
6.4.1 L1 TLB . . . 124
6.4.2 GTLB . . . 125
6.4.3 L2 TLB . . . 125
6.4.4 Sstc Extension . . . 128
6.4.5 Designs for PPA Analysis Selection . . . 129
6.5 Physical Implementation . . . 130
6.5.1 Methodology . . . 130
6.5.2 Results . . . 130
6.6 Discussion . . . 131
6.7 Related Work . . . 133
6.8 Conclusion . . . 134

7 CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation 135
7.1 Introduction . . . 136
7.2 Background . . . 138
7.2.1 A Brief History of Capability-based Protection . . . 138
7.2.2 RISC-V Privileged Specification . . . 138
7.2.3 CHERI ISA in Nutshell . . . 138
7.2.4 RISC-V CVA6 CPU . . . 140
7.3 CVA6 “From Scratch” CHERI Support . . . 140
7.3.1 Capability Library Port to SystemVerilog . . . 141

7.3.2 Capabilities in Pipeline . . . 141
7.3.3 CHERI CSRs, SCRs and CHERI Register File . . . 142
7.3.4 Hybrid Mode Support . . . 142
7.3.5 CHERI Instructions . . . 143
7.3.6 PCC Handling . . . 144
7.3.7 CHERI MMU Extensions . . . 144
7.3.8 CHERI Exception Handling . . . 145
7.3.9 Tagged-Memory Support . . . 145
7.3.10 RVFI-DII Support . . . 145
7.4 Tag Controller . . . 146
7.5 Evaluation . . . 147
7.5.1 Verification and Functional Validation . . . 148
7.5.2 CoreMark Performance Results . . . 149
7.5.3 MiBench Performance Results . . . 150
7.5.4 San-Diego Vision Performance Results . . . 151
7.5.5 MiBench vs San Diego Vision . . . 152
7.5.6 Hardware Overhead . . . 152
7.6 Discussion . . . 153
7.7 Related Work . . . 154
7.8 Conclusion . . . 155

8 Bao-CHERI: A RISC-V Open-Source Hypervisor with CHERI Support 157
8.1 Introduction . . . 158
8.2 System Architecture . . . 160
8.2.1 Platform Requirements . . . 160
8.2.2 Hypervisor Layer . . . 160
8.2.3 VM Layer . . . 161
8.3 Bao-CHERI RISC-V Implementation . . . 162
8.3.1 Boot Process . . . 162
8.3.2 Trap Entry/Exit and Context Switching (CHERI Deltas) . . . 165
8.3.3 Other Modifications . . . 166
8.4 Evaluation . . . 167
8.4.1 Code Size and Memory Footprint . . . 167
8.4.2 Boot Time Overhead . . . 169
8.4.3 Interrupt Latency . . . 170
8.4.4 Performance Overhead . . . 171
8.5 Exploratory Design Directions . . . 173
8.5.1 Compartmentalization of Hypervisor Services: VirtIO Use-Case . . . 173

8.5.2 CHERI as an Alternative to MMU-less Virtualization 175
8.6 Conclusion 176

Bibliography **178**

List of Figures

1 Relationship between research challenges, research questions, and scientific contributions 11

2 Comparison of Type-1, Type-1.5 (hybrid), and Type-2 hypervisors 16
3 Design space of static partitioning hypervisors for mixed-criticality systems 17
4 ACL vs. capability-based access-control models 20
5 CHERI architectural execution modes and deployment configurations 24
6 RISC-V market size by architecture type 26
7 CVA6 core microarchitecture overview 29

8 Taxonomy of memory-safety protection techniques 34
9 AddressSanitizer memory layout 35
10 Tagged memory overview 37
11 Taxonomy of compartmentalization and isolation technologies 46
12 Monolithic vs microkernel architectures 47
13 Comparison of TEE and confidential computing architectures 53

14 A timeline of significant hardware capability systems 77
15 Unified vs. split capability representation schemes 79
16 CHERI hardware capability 128-bit format 84
17 Pyramid chart illustrating CHERI impact across the computing stack 86
18 CHERI hardware capability 256-bit format 88

19 RISC-V privilege levels 114
20 CVA6 core microarchitecture featuring the RISC-V hypervisor extension 117
21 MMU microarchitectural enhancements 120
22 RISC-V nested-MMU walk example for the SV39x4 scheme 121
23 MiBench results for L1 TLB 124
24 MiBench results for GTLB 124
25 MiBench results for L2 TLB multi-page size 125

26 MiBench results for L2 TLB associativity 126
27 MiBench results for Sstc extension 127
28 Summary of MiBench functional performance results 128
29 San Diego Vision Benchmark functional performance results for PPA analysis 129
30 Baseline CVA6 layout (0.75mmx0.65mm) 130
31 Area and power results 130
32 Energy efficiency results 132

33 CVA6 CHERI capability 128-bit compressed format 139
34 CVA6 pipeline microarchitecture featuring the RISC-V CHERI extension 140
35 CVA6 CHERI capability in-pipeline formats 142
36 Tag Controller microarchitecture design high-level overview 146
37 TestRIG verification setup overview 149
38 MiBench performance overhead results for the CVA6H-CHERI 151
39 San Diego Vision performance overhead results for the CVA6H-CHERI 151

40 Tri-core Bao platform with CHERI-enabled and legacy VMs 161
41 Bao-CHERI boot sequence 163
42 MiBench performance overhead results for Bao-CHERI 172
43 Architecture overview of CHERI-enabled VirtIO design in Bao 173

List of Tables

1 Summary of isolation technologies for embedded systems 5
2 Comparison of access-control and capability-based protection 6

3 Summary of RISC-V privilege specification 28

4 Examples of spatial and temporal memory-safety CVEs (2024–2026). 33
5 Summary of memory safety enforcement and mitigation techniques 41
6 Summary of memory-safety hardening techniques 45
7 Summary of microarchitectural and memory-system isolation techniques 51
8 Comparison of memory-safety techniques and positioning of CHERI 56
9 Comparison of compartmentalization and isolation techniques for MCS 59

10 Hardware capability-based systems 78
11 Software capability-based OSs 82
12 Basic principles of the CHERI ISA 84
13 Public implementations of CHERI CPUs 90
14 List of CHERI-aware OS, RTOS, and Microkernels 95
15 CHERI capability revocation techniques 99
16 CHERI emulators, tools, and debugger 102

17 Table of abbreviations 113
18 Hypervisor extension features implemented in the CVA6 core 118
19 Design space exploration configurations 123
20 L2 TLB design space exploration configurations 126
21 SSTC design space exploration configurations 128
22 7 selected configurations for PPA analysis in GF22nm 131
23 CVA6-like RISC-V Cores with hypervisor extension support 133

24 CHERI Capability RTL library interface in SystemVerilog 141

25 CHERI extension features implemented in the CVA6 core 143
26 CoreMark/MHz results for baseline CVA6 and CHERI-aware CVA6 150
27 Post-synthesis FPGA resource overhead of CVA6H-CHERI 152
28 Publicly available CHERI-enabled RISC-V CPU implementations 154

29 Instruction and CSR mapping required for transitioning to CHERI pure-capability execution 165
30 Source-level code changes by directory and language 168
31 Binary section sizes by directory 169
32 Per-vCPU runtime memory overhead 169
33 Boot time overhead 170
34 Interrupt latency overhead 170
35 Comparison of isolation mechanisms for embedded virtualization 176

Acronyms

ABI	Application Binary Interface
ACL	Access Control List
AI	Artificial Intelligence
AIA	Advanced Interrupt Architecture
API	Application Programming Interface
APL	Access Protection List
APU	Application Processor Unit
ASan	AddressSanitizer
ASID	Address Space Identifier
ASLR	Address Space Layout Randomization
AXI	Advanced eXtensible Interface
BHT	Branch History Table
BRAM	Block Random Access Memory
BTB	Branch Target Buffer
BTI	Branch Target Identification
CAGR	Compound Annual Growth Rate
CAMB	Cloud Attestables on Morello Boards
CC	Confidential Computing
CCA	Confidential Computing Architecture
CDT	Capability Derivation Tree
CET	Control-Flow Enforcement Technology
CFI	Control-Flow Integrity
CHERI	Capability Hardware Enhanced RISC Instructions
CHERIoT	Capability Hardware Enhanced RISC Instructions for the Internet of Things

CID Compartment Identifier
CLINT Core-Local Interrupt Controller
CLU Cheri Logic Unit
CoVE Confidential VM Extension
CPI Code-Pointer Integrity
CPS Cyber-Physical System
CPU Central Processing Unit
CSA Clang Static Analyzer
CSR Control and Status Register
CVE Common Vulnerabilities and Exposures
CVM Confidential Virtual Machine
cVM Capability Virtual Machine
CVSS Common Vulnerability Scoring System
CWE Common Weakness Enumeration

DCache Data Cache
DDC Default Data Capability
DFI Data-Flow Integrity
DMA Direct Memory Access
DRAM Dynamic Random Access Memory
DSbD Digital Security by Design
DSE Design Space Exploration

ECC Error-Correcting Code
ECU Electronic Control Unit

FF Flip-Flop
FIFO First-In, First-Out
FPGA Field-Programmable Gate Array
FPU Floating-Point Unit
FVP Fixed Virtual Platform

GC Garbage Collection
GDB GNU Debugger
GPOS General-Purpose Operating System
GPU Graphics Processing Unit
GTLB G-Stage Translation Lookaside Buffer

GUI Graphical User Interface

HOL Higher Order Logic
HPC High-Performance Computing
HPM Hardware Performance Monitor

I/O Input/Output
ICache Instruction Cache
ICS Industrial Control System
IDS Intrusion Detection System
IOMMU Input/Output Memory Management Unit
IOPMP Input/Output Physical Memory Protection
IoT Internet of Things
IP Intellectual Property
IPC Inter-Process Communication
IPI Inter-Process Interrupt
IRQ Interrupt Request
ISA Instruction Set Architecture
ISCF Industrial Strategy Challenge Fund
ISO International Organization for Standardization

JIT Just-in-Time
JOP Jump-Oriented Programming
JSC JavaScriptCore
JVM Java Virtual Machine

KASLR Kernel Address Space Layout Randomization
KVM Kernel-based Virtual Machine

LLC Last-Level Cache
LSE Local State Encapsulation
LUT Lookup Table

MAC Message Authentication Code
MCS Mixed-Criticality Systems
MCU Microcontroller Unit
MIPS Microprocessor without Interlocked Pipeline Stages

ML Machine Learning
MMIO Memory-Mapped Input/Output
MMU Memory Management Unit
MPU Memory Protection Unit
MSI Message Signaled Interrupts
MTE Memory Tagging Extension

NAPOT Naturally Aligned Power-of-Two
NIC Network Interface Card
NPU Neural Processing Unit

OoO Out-of-Order
OS Operating System
OTA Over-the-Air

PAC Pointer Authentication Code
PC Program Counter
PCB Printed Circuit Board
PCC Program Counter Capability
PIE Pointer Integrity
PLIC Platform-Level Interrupt Controller
PMP Physical Memory Protection
PPA Power, Performance, and Area
PTE Page Table Entry
PTW Page Table Walker

QEMU Quick Emulator

RAM Random-Access Memory
RAS Return Address Stack
RISC Reduced Instruction Set Computer
RME Realm Management Extension
ROP Return-Oriented Programming
RTL Register-Transfer Level
RTOS Real-Time Operating System

SAE Society of Automotive Engineers

SBI Supervisor Binary Interface
SCR Special Capability Register
SEV Secure Encrypted Virtualization
SGX Software Guard Extensions
SIMT Single Instruction Multiple Threads
SMT Satisfiability Modulo Theories
SoC System on Chip
SPARC Scalable Processor ARChitecture
SPH Static Partitioning Hypervisor
SPI Serial Peripheral Interface
SPMP Supervisor Physical Memory Protection
SWaP-C Size, Weight, Power, and Cost

TARA Threat Analysis and Risk Assessment
TCB Trusted Computing Base
TCP/IP Transmission Control Protocol / Internet Protocol
TCU Telematics Control Unit
TEE Trusted Execution Environment
TLB Translation Lookaside Buffer
TPU Tensor Processing Unit

UAF Use-After-Free
UART Universal Asynchronous Receiver-Transmitter
UAV Unmanned Aerial Vehicle
UK United Kingdom
UMC User Management Component

VE Virtualization Extensions
VM Virtual Machine
VMID Virtual-Machine Identifier
VMM Virtual Machine Monitor

WBCF Well-Bracketed Control Flow

XOM Execute-Only Memory

Part I

Dissertation Summary

1 Introduction

Modern embedded computing systems are transitioning from single-purpose devices to complex multicore heterogeneous platforms that integrate multiple software stacks of different criticality levels on shared hardware, i.e., Mixed-Criticality Systems (MCS). As these software stacks continue to grow, predominantly written in C/C++, and adopt increasing levels of connectivity, the attack surface expands in proportion, exposing systems to both external adversaries and latent threats, such as memory-safety vulnerabilities. Security has therefore become a fundamental design concern in contemporary embedded systems. However, traditional mitigation and hardening mechanisms struggle to provide strong security guarantees without imposing prohibitive performance costs or compatibility overhead, and generally lack the fine-grained compartmentalization required to constrain the impact of a successful attack. Consequently, hardware support for isolation and memory safety has become central in embedded system design. Among emerging approaches, capability-based protection stands out for following the security-by-design philosophy, enforcing deterministic memory safety and architectural compartmentalization directly in hardware rather than relying on software layers or incremental defenses that target only specific classes of attacks. In particular, the Capability Hardware Enhanced RISC Instructions (CHERI) architecture demonstrates how capabilities can be integrated into commodity Instruction Set Architectures (ISAs) to provide strong memory safety together with lightweight, fine-grained compartmentalization, while preserving compatibility with legacy C/C++ codebases. This positions CHERI as a promising foundation for building safe, secure, and trustworthy embedded systems. Nevertheless, the impact of adopting CHERI within mixed-criticality embedded systems remains unexplored in the literature.

1.1 Motivation

1.1.1 The Evolution of Embedded Systems

Embedded systems play a critical role in safety-critical domains such as transportation, automotive, space, and industrial systems, where a single failure can lead to hazardous consequences and a potential loss of life [1–5]. In recent years, these platforms have evolved from distributed single-purpose controllers to

complex multicore and heterogeneous System on Chips (SoCs) driven by the growing need to consolidate safety-critical and non-critical workloads onto the same hardware and therefore reduce Size, Weight, Power, and Cost (SWaP-C) and certification cost [6–8]. Although consolidation offers significant economic and performance benefits, it also poses new challenges in providing strong isolation due to the sharing of hardware resources between different domains of execution [9, 10]. A critical software fault in one component may affect the entire system unless the isolation mechanisms are sufficiently robust and fine-grained [11]. For example, in the automotive domain, there is the emergence of software-defined vehicles, where the hardware remains relatively static and the functionalities increase through software updates and upgrades rather than hardware [12]. Features such as Over-the-Air (OTA) updates, remote diagnostics, and cloud-connected services demand widespread connectivity, turning previously closed embedded systems into highly interoperable cyber-physical platforms. As a result, the software stacks continue to grow and the attack surface increases dramatically, creating new opportunities for adversaries to exploit vulnerabilities and undermine the safety properties of these systems [13, 14].

The safety community has traditionally focused on demonstrating sufficient independence and a low probability of dependent failures between safety functions [6, 15–17]. However, the increasing digitalization has exposed embedded platforms to external threats. Modern attacks have demonstrated that compromising a non-critical software stack can endanger safety-critical behavior by exploiting memory-safety bugs or bypassing isolation mechanisms [18, 19]. As a result, robust isolation and memory safety are now prerequisites rather than optional add-ons in safety-critical industries [20, 21]. This evolution toward high-integration architectures calls for security-by-design approaches capable of ensuring that faults or compromises in non-critical software cannot endanger safety-critical functions. However, most existing defenses follow a reactive trajectory, i.e., new vulnerability classes are discovered and subsequently mitigated through incremental or ad hoc countermeasures, perpetuating an ongoing cycle of attacks and patches. Critically, by the time vulnerabilities are disclosed and patches deployed, sensitive data are often already exfiltrated, and the resulting damage cannot be undone. Empirical studies confirm this risk, showing that zero-day vulnerabilities are frequently weaponized within days, or even hours, of disclosure [22], while many breaches remain undetected for months before containment [23]. Ultimately, this arms race structurally favors attackers, i.e., the continuous emergence of zero-day vulnerabilities, combined with the inherently reactive nature of most mitigation strategies, leaves defenders unable to keep pace with the scale and sophistication of modern attacks. To break this cycle, the community must explore architectural solutions that enforce memory safety and isolation from the ground up.

1.1.2 Memory Safety for Embedded Systems

Memory safety remains one of the most persistent and critical challenges in modern computing systems. Non-memory-safe languages, such as C and C++, remain prevalent in safety-critical and real-time domains, valued for their low-level control, deterministic performance, and efficient resource usage, all essential requirements in embedded environments. However, this flexibility produces a significant risk, as

developers must manually manage memory lifecycles, exposing software to spatial and temporal safety violations such as buffer overflows, dangling pointers, and Use-After-Free (UAF) vulnerabilities. Empirical data from industry leaders, including Microsoft, Google, and Apple, has consistently shown that the majority of security patches in large software systems are related to memory safety violations [24–26].

A range of approaches has been developed to address these challenges. Static analysis tools (e.g., Cppcheck [27] and Coverity [28]) identify potential bugs early in the development cycle by analyzing the source code. Although useful during the early stages of the software development cycles [28, 29], these tools often produce false positives. Compiler-based instrumentation offers runtime detection with minimal code changes but often introduces high execution and memory overheads, making it unsuitable for production environments [30]. Hardware-assisted sanitizers (e.g., WatchdogLite [31]) leverage hardware features to accelerate compiler-inserted memory-safety checks, while approaches such as HWASan [32] rely on lightweight hardware support for metadata propagation. Nevertheless, both approaches require specialized toolchains and sometimes custom hardware features, which limit their deployability in practice.

More recently, hardware-enforced memory safety techniques have gained traction. Examples include tagged memory (e.g., Arm MTE [33] and RISC-V Zimt extension [34]), cryptographic security extensions (PACMem [35], CryptSan [36]), and capability-based architectures such as CHERI [37]. These mechanisms embed metadata into the hardware itself, enabling more efficient and predictable enforcement than purely software-based approaches. However, they require substantial integration throughout the software stack, from compiler to the Operating System (OS), and have limited availability on commercial embedded platforms.

In parallel, memory-safe languages such as Rust [38] and SPARK/Ada [39] promote safety through strong type systems and ownership semantics. While promising, these languages face practical limitations: (i) low-level tasks (e.g., memory-mapped Input/Output (I/O), interrupt handling) often require unsafe constructs that bypass safety guarantees and (ii) rewriting legacy C/C++ codebases disrupts certification paths and entails high engineering costs. Therefore, companies follow hybrid strategies that focus on gradually adopting safe language components while maintaining interoperability with legacy C/C++ code, rather than attempting a one-step full migration [40].

All in all, the embedded systems landscape continues to rely heavily on C/C++, with memory safety only partially addressed via static analysis [27, 28, 41, 42], sanitizers [43–45], safe-language modules [38, 39], hardware support [33, 46–49], and strict coding standards (e.g., MISRA-C). Despite decades of research and tooling, practical memory safety remains elusive, particularly in mixed-criticality embedded environments, where violations can have catastrophic consequences. This gap highlights the need for architectural solutions that enforce memory safety deterministically, without sacrificing performance, real-time requirements, or compatibility with existing embedded systems stacks.

Technology	Granularity	Limitations
Virtualization	VM-level	Strong inter-VM isolation, but fine-grained intra-VM or intra-hypervisor compartmentalization requires additional mechanisms.
Microkernels	Component-level	Requires explicit system decomposition, careful interface design, and non-trivial porting effort.
Enclaves / TEEs	Enclave-level	Limited scope. Targets application or trusted-service isolation and relies on trusted runtime/firmware components.
SoC-wide isolation	Domain-level	Coarse-grained protection domains across CPUs, memory, and devices. Requires platform-wide hardware and firmware support.

Table 1: Summary of system-level isolation technologies commonly used in embedded systems.

1.1.3 Isolation Technologies for Embedded Systems

The growing number of vulnerabilities and the expanding size of embedded software stacks have led academia and industry to explore isolation technologies for compartmentalizing functionality and limiting privilege escalation. At the hardware level, isolation is traditionally grounded in enforcement substrates, such as privilege levels, address translation through Memory Management Units (MMUs), and region-based protection through Memory Protection Units (MPUs). These mechanisms provide isolation at different granularities: privilege levels separate execution modes, MMUs isolate address spaces, and MPUs enforce protection over memory regions, often in platforms where full address translation is unavailable or undesirable due to timing variability from Translation Lookaside Buffer (TLB) misses. However, they remain coarse-grained building blocks, as they do not directly provide fine-grained compartmentalization within a process, a Virtual Machine (VM), or the hypervisor itself. Building on these hardware substrates, system-level isolation technologies provide different trade-offs in terms of granularity and complexity, as summarized in Table 1. For instance, virtualization encapsulates VMs, allowing multiple workloads to be consolidated, preserving compatibility with legacy software stacks, and enforcing strict isolation between domains [50]. Although hypervisors provide effective inter-VM isolation, they do not inherently provide fine-grained isolation within a VM or within the hypervisor itself. Consequently, a compromise of a high-privilege component, such as a monolithic Real-Time Operating System (RTOS), an OS service, or a hypervisor component, may still endanger the broader system. Microkernels such as seL4 [51] and Fiasco.OC [52, 53] address this limitation by decomposing systems into isolated components that communicate through explicit kernel-mediated interfaces, often based on capability-like access-control mechanisms. This design can reduce the Trusted Computing Base (TCB), limit fault propagation, and improve fault isolation. Hypervisors provide isolation by virtualizing the platform so that multiple OSs can run concurrently, improving workload consolidation, maintaining backward compatibility with legacy stacks, and ensuring strong separation between domains [50].

Over the past few years, commodity ISAs have gradually incorporated hardware-enforced isolation extensions [54–57]. For example, Arm TrustZone [58] partitions execution into secure and non-secure worlds, enclave-based confidential computing (e.g., Intel Software Guard Extensions (SGX) [59], RISC-V Keystone [60]) protects code and data from untrusted OSs to run trusted services (e.g., cryptographic

operations), and hardware virtualization extensions (e.g., RISC-V hypervisor extension [56] and Arm Virtualization Extensions (VE) [55]) provide architectural mechanisms that enable efficient hypervisor implementations. Although powerful, these mechanisms have limitations in scalability, composability, and attack-surface reduction, and several deployments have been affected by vulnerabilities arising from complex trusted software stacks, shared resources, or incomplete mediation [58, 61, 62]. For instance, virtualization provides consolidation at the cost of shared-resource leakage and having the hypervisor as a single point of failure. Static-partitioning virtualization (e.g., Bao hypervisor) improves determinism, but restricts dynamic software composition. As embedded platforms adopt service-oriented architectures and rely increasingly on OTA updates, vendor libraries, trusted services, and third-party modules, the assumption of mutual trust within a protection domain becomes untenable. Modern systems require internal isolation not only between OSs or VMs, but within them. As a response, recent mechanisms such as Arm Confidential Computing Architecture (CCA), Realm Management Extension (RME) [63], RISC-V Confidential VM Extension (CoVE) [64], and RISC-V WorldGuard [65] have focused on providing isolation enforced throughout the SoC, including accelerators, Direct Memory Access (DMA) engines, Network Interface Cards (NICs), and memory controllers. By checking access rights in hardware rather than relying solely on software policy, these approaches aim to eliminate trust bottlenecks and overcome the scalability limits of coarse-grained isolation.

Despite major progress, current isolation technologies (e.g., virtualization) do not inherently prevent intra-process memory corruption, and large monolithic workloads remain incompatible with fine-grained compartmentalization. Consequently, isolation and memory safety must be addressed jointly: isolation limits the blast radius of compromise, while memory safety reduces the likelihood of compromise. This convergence naturally motivates capability-based protection, which offers architectural support for deterministic memory safety and fine-grained compartmentalization within a shared address space.

1.1.4 Hardware Capability-based Protection: Security-by-Design

Approach	Protection Model	Key Limitation	Examples
ACL-based systems	Identity-based access control	Relies on global policy checks and is susceptible to confused-deputy attacks	POSIX Access Control Lists (ACLs) on Linux
Software capabilities	Unforgeable software references	Requires a trusted kernel and increases system complexity and maintenance effort	seL4, KeyKOS, EROS
Hardware capabilities	Hardware-enforced capabilities	Requires software-stack changes, incurs hardware overhead, and faces capability-revocation challenges	System/38, M-Machine

Table 2: Comparison of access-control and capability-based protection models.

Hardware-enforced capability architectures provide two key security primitives: (i) deterministic memory safety through pointer-level bounds enforcement and (ii) direct hardware support for fine-grained compartmentalization semantics within the ISA, rather than relying on software-managed isolation. Although the concept of capabilities dates back to the seminal work of Dennis and Van Horn in the 1960s [66], both software and hardware capability systems have been explored to realize least-privilege execution

models [67–69]. Capability-based protection differs fundamentally from ACLs [70], which define access rights externally to objects and require the system to maintain and consult global access control tables at runtime. This results in coarse-grained, centralized decision-making that often fails to scale efficiently in dynamic, multi-tenant, or fine-grained isolation scenarios and is susceptible to confused deputy attacks due to ambient authority [71]. In contrast, capabilities are decentralized, unforgeable tokens of authority that are held by the subjects themselves. They encode not only access rights but also the address, bounds, and permissions of the object, enforced directly in hardware. As such, capability architectures avoid ambient authority and ensure that access is always intentional, explicit, and least-privilege by construction [71]. Table 2 highlights the fundamental differences in protection models and key limitations between traditional access-control mechanisms and capability-based protection models.

Software capability systems, such as those implemented in microkernels like seL4 [51], KeyKOS [72], and EROS [68], represent capabilities as logical objects managed by the kernel. However, their enforcement is entirely dependent on the correctness of the kernel, which is difficult to guarantee in large systems or costly to formally verify [73]. Moreover, the microkernel design model shifts significant complexity to user space, requiring developers to decompose system services into isolated components and explicitly manage Inter-Process Communication (IPC) and capability propagation using kernel-provided interfaces, thus increasing the effort required to build and maintain complete systems. In contrast, hardware capability systems such as IBM System/38 [74–76], Hydra [67], M-Machine [77], and SAFE [46] pushed enforcement to the Central Processing Unit (CPU) itself. These architectures embedded capabilities directly into memory and processor registers, enforcing bounds, permissions, and provenance checks in hardware at every memory access. This approach provides deterministic memory safety and fine-grained authority control, enabling true least-privilege execution within a single address space. Despite their strong security properties, early hardware capability systems did not achieve mainstream adoption due to several factors: (i) complexity to manage and revoke capabilities [78], (ii) market preference for performance over security [79, 80], (iii) the cost of silicon at the time, (iv) incompatibility with legacy C/C++ software ecosystems [79, 81], and (v) the need for deep toolchain and OS changes. As embedded platforms increasingly consolidate complex update-driven software stacks, capability-based protection offers a scalable and robust alternative to traditional “detect-and-patch” security models.

1.1.5 CHERI for Next-Generation Secure Systems

CHERI [37] is a modern realization of hardware-enforced capability-based protection for contemporary ISAs. At its core, CHERI replaces raw pointers with architecturally defined capabilities, *unforgeable tokens of authority* that pair an address with metadata describing the memory region it can access (bounds), permitted operations (permissions), an optional object type (used to support abstract software-defined objects), and a hardware-maintained validity tag that ensures integrity. These metadata invariants are enforced on every memory access and control-flow transfer, providing deterministic spatial memory safety

and control-flow integrity. Moreover, capabilities follow strict semantics enforced by hardware: (i) capabilities must be derived from existing valid capabilities (valid provenance) and (ii) derived capabilities must confer equal or reduced authority (monotonicity). These principles guarantee that access is always explicit, intentional, and least-privilege by construction.

A key strength that distinguishes CHERI from previous attempts to define a hardware-enforced capability system is its incremental deployability [82]. Rather than requiring a complete rewrite of existing C/C++ software stacks, CHERI enables hybrid execution, allowing unmodified legacy code to coexist with selectively adapted components that adopt capabilities. This is supported by compiler infrastructure (e.g., Clang/LLVM), hybrid Application Binary Interfaces (ABIs), OSs, and mixed-capability runtimes, providing a realistic and practical migration path toward adoption. Beyond memory safety, CHERI also supports lightweight and fine-grained compartmentalization within a single address space, allowing strong isolation of system components such as device drivers, shared libraries, OS services, and user-level modules.

In the last decade, CHERI has evolved from a research prototype [37] to a growing industrial ecosystem [83, 84]. The Arm Morello platform, developed under the United Kingdom (UK) initiative Digital Security by Design (DSbD) [85], demonstrated the viability of CHERI for server-class systems and industrial-scale evaluation. In parallel, the Sonata platform [86] and the Microsoft Capability Hardware Enhanced RISC Instructions for the Internet of Things (CHERIoT) project [87] have adapted CHERI for deeply embedded microcontrollers with tight memory constraints. Codasip has integrated CHERI into commercial RISC-V Intellectual Property (IP) cores (e.g., A730) [84]. Several software-support efforts have followed, including compilers [88], runtimes [82, 89–94], emulators [95, 96], tools [97], firmware (e.g., OpenSBI) [98], OSs (e.g., CheriBSD [89, 99], CheriLinux [100, 101]), microkernels (e.g., sel4) [102], initial support for capability-aware RTOSs (e.g., Cheri-Zephyr) [103], compartmentalized `libc` implementations [104, 105], and several applications. A recent milestone was the formation of the CHERI Alliance in late 2024 [106], a consortium that included Arm, Google, Codasip and lowRISC, with the aim of coordinating the global adoption of CHERI. Complementing this effort, the RISC-V community has launched the `Zcheri` standardization initiative, which defines a common set of CHERI ISA extensions for 64-bit (RV64) and 32-bit (RV32) profiles [107]. `Zcheri` represents a key step toward the widespread adoption of CHERI technology in the RISC-V ecosystem.

Despite these advances, several open challenges must be addressed before CHERI can achieve widespread adoption, particularly in the domain of mixed-criticality embedded systems. Most CHERI research to date has focused on server/cloud-class platforms or deeply embedded microcontrollers, with significant progress in CPU extensions, hardware prototypes, tagged-memory support, compiler toolchains, OSs, and RTOSs. A particularly active area of investigation is capability revocation for temporal safety, one of the most complex and crucial challenges in capability-based systems. However, the design space for intermediate-scale embedded platforms with mixed-criticality and safety requirements and integration with legacy technologies already largely deployed in the industrial settings (e.g., virtualization) remains largely underexplored. Bridging this gap is essential to fully realize the potential of CHERI technology as a foundational architecture for MCS.

1.2 Contributions

1.2.1 Research Challenges

Despite the growing maturity of the CHERI technology, several open challenges remain in the context of secure mixed-criticality and virtualized embedded systems. This dissertation focuses on addressing three core research challenges:

RC1. Scattered and Fragmented Information on CHERI Technology. Despite growing academic and industrial interest, knowledge about the CHERI ecosystem remains fragmented across dispersed sources, including academic publications, project reports, hardware manuals, open-source repositories, mailing lists, conference talks, and white papers. Currently, there is no comprehensive and systematized reference that compiles critical knowledge about CHERI architecture, microarchitectural implementation techniques, compiler and toolchain support, OS integration, runtime support, security guarantees, and empirical evaluation data. Consequently, this fragmentation hinders widespread adoption of the technology, including understanding design trade-offs, comparing existing implementations, and identifying open research challenges. Therefore, there is a clear need for a comprehensive survey that systematically synthesizes the state-of-the-art in CHERI, establishes a common terminology, and provides a reliable reference point for future CHERI research and engineering efforts.

RC2. Cost and Design Implications of Integrating CHERI and Virtualization in RISC-V Processors. Despite ongoing efforts in both academia and industry, existing RISC-V implementations support either CHERI (e.g., `CHERI-Flute`, `Cheri-Tooba`, Codasip X730) or the hypervisor extension (e.g., `Rocket Chip`), but not both within a single processor design. Consequently, the interplay between capability-based protection and virtualization mechanisms remains largely unexplored at both the architectural and microarchitectural levels. This motivates the need to investigate the cost, design trade-offs, and implications of integrating CHERI with virtualization support on a RISC-V processor. This platform will enable a deeper understanding of their interaction at the hardware level and provide a foundation for exploring higher-level protection models and their integration with virtualization software stacks, including Static Partitioning Hypervisor (SPH). Ultimately, this enables a systematic evaluation of CHERI and supports its practical deployment in virtualization-aware systems.

RC3. Limited Understanding of CHERI Integration into Static Partitioning Hypervisors. Although CHERI provides hardware-enforced memory safety and supports fine-grained compartmentalization, its integration into hypervisor-based isolation remains largely underexplored, particularly in SPHs targeting mixed-criticality embedded systems. In such systems, hypervisors (e.g., Bao) enforce strong isolation through the static allocation of resources across domains or VMs. However, there is limited understanding of the engineering effort required to extend these hypervisors with CHERI support, including

modifications to boot sequence, context management, and interaction with capability-aware hardware. Moreover, the potential benefits of such integration, in terms of improved security, fault isolation, and support for least-privilege system decomposition, have not been systematically evaluated. This lack of understanding hinders a clear comparison between capability-based isolation and traditional approaches based on MMUs and MPUs. Therefore, it is necessary to investigate the integration effort, system-level costs, and security benefits of incorporating CHERI into SPHs, enabling a systematic assessment of its impact on hypervisor design.

1.2.2 Research Questions

Based on the research challenges identified above, this dissertation investigates how CHERI technology can be integrated with virtualization support on RISC-V-based platforms for mixed-criticality embedded systems. The goal is to understand the state of the CHERI ecosystem, the architectural and microarchitectural implications of combining CHERI with virtualization, and the feasibility of supporting capability-aware virtualization in a SPH. This dissertation is guided by the following research questions:

RQ1: What is the current state, maturity, and research landscape of CHERI technology across the hardware-software stack?

RQ2: What are the design trade-offs associated with implementing and optimizing the RISC-V hypervisor extension in an open-source processor for mixed-criticality systems?

RQ3: What are the hardware costs, performance implications, and microarchitectural impacts of integrating CHERI support into a virtualization-enabled RISC-V processor for mixed-criticality embedded systems?

RQ4: What is the engineering effort required to integrate CHERI technology into a SPH, such as Bao, and what are the associated system-level costs and security benefits?

To address these questions, this dissertation follows a stack-oriented methodology. **RQ1** is addressed through a systematization of the CHERI ecosystem, covering the full system stack. **RQ2** is addressed by implementing and evaluating the RISC-V hypervisor extension in the open-source CVA6 processor, including virtualization-related microarchitectural optimizations. **RQ3** is addressed by extending this virtualization-enabled processor with CHERI support and evaluating its hardware and performance costs. **RQ4** is addressed by integrating CHERI into the Bao SPH and assessing the resulting engineering effort, system-level overheads, and opportunities for capability-aware virtualization. The following subsection summarizes the contributions derived from these research questions.

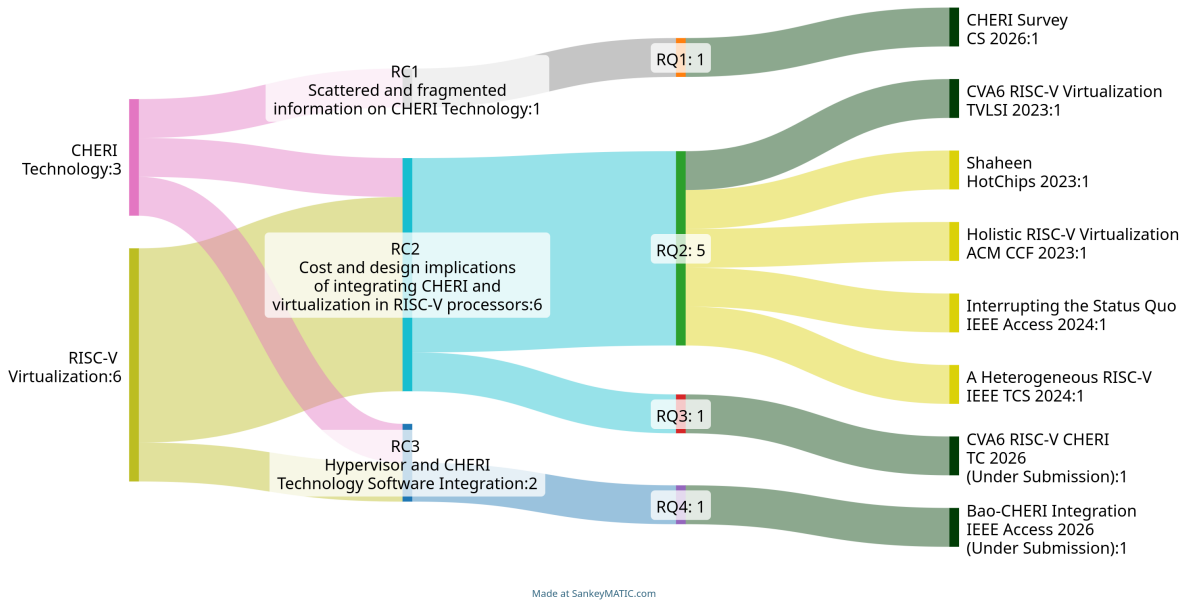


Figure 1: Relationship between research challenges, research questions, and scientific contributions.

1.2.3 Main Contributions

Figure 1 summarizes the core scientific contributions of this dissertation and how they relate to the research challenges and questions outlined in this chapter. The four primary contributions developed during the course of this work are highlighted in dark green: (i) a comprehensive survey on CHERI technology, (ii) implementation of the hypervisor extension on the CVA6 core with virtualization-oriented microarchitectural enhancements, (iii) a full-stack reference platform for CHERI-aware virtualization on RISC-V using CVA6, and (iv) the addition of CHERI support to the Bao hypervisor. These contributions are presented in detail in Chapters 5, 6, 7, and 8, respectively, and have led to the publication of two peer-reviewed journal articles, with two additional manuscripts currently under review. Secondary contributions are highlighted in yellow and include collaborative publications that fall outside the main scope of this document, specifically "Shaheen: An Open, Secure, and Scalable RV64 SoC for Autonomous Nano-UAVs", "A Heterogeneous RISC-V Based SoC for Secure Nano-UAV Navigation", "Interrupting the Status Quo: A First Glance at the RISC-V Advanced Interrupt Architecture (AIA)", and "Holistic RISC-V Virtualization: CVA6-based SoC " [108–110].

Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology. In this work, we presented the first comprehensive systematization of the CHERI technology, spanning the hardware and software stack [111]. This survey provided a structured overview of CHERI architecture variants (e.g., CHERI-MIPS, CHERI-RISC-V, Arm Morello), microarchitectural enforcement mechanisms, compiler and toolchain support, OS and runtime integration, and formal security guarantees. We consolidated and analyzed a highly fragmented body of knowledge drawn from academic research, industrial reports, open-source repositories, and ongoing standardization efforts such as the `Zcheri` RISC-V extensions. Beyond technology mapping, the survey also identified the core principles of CHERI, such as

unforgeable capabilities, spatial memory safety, and fine-grained compartmentalization enforcement, and provided a comparison with other hardware capability systems. We reviewed real-world and experimental use cases of CHERI, from secure compartmentalization in OS kernels and userspace to its adoption in Internet of Things (IoT) and real-time embedded systems (e.g., CHERIoT). The survey concluded with a discussion on future research directions, including integration with virtualization and applicability to mixed-criticality systems, heterogeneous computing, large-scale applications, Trusted Execution Environments (TEEs), side-channel attacks, and system-level isolation. This work established a common terminology and reference point for the research community and industry to understand CHERI technology and its impact on the ecosystem. The paper was accepted in ACM Computing Surveys in collaboration with Minerva Systems and the University of Reggio Emilia and is detailed in Chapter 5.

CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration. This work presented the first open-source implementation of version 1.0 of the RISC-V hypervisor extension in the CVA6 core [112]. Our contributions were multifold and encompassed full architectural support for virtualization, microarchitectural enhancements, and physical implementation analysis. We extended CVA6 with key virtualization features, including second-stage address translation, virtual privilege levels, hypervisor instructions, guest and hypervisor Control and Status Registers (CSRs), interrupt and exception handling, and trap redirection. To mitigate the performance overhead of second-stage address translation, we introduced an L2 TLB and a G-Stage Translation Lookaside Buffer (GTLB) to accelerate the Page Table Walker (PTW), significantly reducing TLB miss penalties and translation latency. Furthermore, we conducted a Design Space Exploration (DSE) of multiple design variants on a Field-Programmable Gate Array (FPGA) (Genesys 2 board) to identify a preferred configuration in terms of performance. We complemented this analysis with post-layout simulations using GlobalFoundries 22nm FDX technology to assess Power, Performance, and Area (PPA) and determine the optimal design point. When evaluated using MiBench workloads running on Linux atop the Bao hypervisor, the preferred configuration achieved an average speedup of 12.5%, with peaks of up to 16%, compared to a baseline design without microarchitectural virtualization optimizations. These gains were achieved with minimal overhead, incurring only a 0.78% increase in area and a 0.33% increase in power consumption. All artifacts from this work have been upstreamed to the CVA6 mainline, maintained by the OpenHW Group, and are currently used in several European projects and industrial settings, including REBECCA, ISOLDE, and RIGOLETTO, as well as by companies such as Thales. This established our design as a foundational reference for future RISC-V research and the adoption of virtualization in embedded cores. The work was published in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* in collaboration with ETH Zurich and the PULP team, and is detailed in Chapter 6.

CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation. This work presented the first integration of the CHERI RISC-V ISA extension into the open-source CVA6 processor [113]. Our contributions spanned both the architectural and the microarchitectural domains. Architecturally, we extended CVA6 with full CHERI capability support, including a capability-aware datapath, merged capability

registers, support for hybrid and pure-capability execution modes, and decoding logic for CHERI-specific instructions (e.g., capability-aware load/store). We also introduced support for CHERI exception handling and capability system instructions. At the microarchitectural level, we developed a SystemVerilog-based capability manipulation library responsible for operations such as capability creation, sealing/unsealing, bounds checks, compression, and decompression. This library has been formally verified against the original Bluespec implementation from the University of Cambridge. To support the CHERI tagged memory model, we extended the cache hierarchy and memory subsystem by implementing a dedicated tag memory controller, a tag cache for fast access, and a backing store in Dynamic Random Access Memory (DRAM) using a 1-bit tag per 128-bit word layout. To ensure functional correctness and compliance with CHERI ISA, we verified our design using the TestRIG framework, the de facto standard for CHERI verification, which performs random instruction injection and trace-based comparison against the formal Sail model. For evaluation, we deployed our CHERI-enabled CVA6 on FPGA (Genesys 2 board) and used the San Diego Vision simulation framework. We ran CheriBSD and exercised real-world embedded workloads from the MiBench suite to validate system functionality and benchmark performance under representative scenarios. We observed a performance slowdown of approximately 20% on average when comparing legacy and pure-capability code on CheriBSD, alongside a 50% increase in hardware resources. Our implementation was initially based on the CHERI v9 specification and has been incrementally updated to align with the official RISC-V `Zcheri` specification currently under ratification. This work was carried out in collaboration with the University of Cambridge, and the core has become one of the reference platforms supporting the standardization process. The paper was submitted to the IEEE Transactions on Computers in collaboration with Minerva Systems and the University of Reggio Emilia and is detailed in Chapter 7.

Bao-CHERI: A RISC-V Open-Source Hypervisor with CHERI Support. This work presented the first integration of CHERI into the open-source Bao hypervisor targeting the RISC-V architecture. Our contributions spanned the system-level integration of CHERI within a virtualization stack, the implementation of a capability-aware hypervisor execution model, and the empirical evaluation of its overall impact. At the implementation level, we extended Bao to execute in pure-capability mode and support capability-aware VMs. Concretely, this required modifications to the boot process, trap entry and exit, context switching, memory management, and Memory-Mapped Input/Output (MMIO) device access. For example, we introduced a CHERI-aware trap handling path that saved and restored capability state, switched to bounded hypervisor-private stacks, and preserved capability-safe privilege transitions between guest and hypervisor execution. To evaluate the integration, we conducted a systematic analysis of engineering effort and runtime overhead. Static analysis revealed a code growth of approximately 17%, with binary size increases of up to 30.2% in the `.text` section and 59.4% in `.bss`. Runtime evaluation showed a boot-time overhead of 20.09% and an interrupt-latency overhead of 13.43%, remaining within acceptable bounds for embedded use cases. Beyond empirical evaluation, we also outlined forward-looking design directions enabled by this integration, including CHERI-protected hypervisor services (e.g., VirtIO backend isolation) and MMU-less virtualization using capabilities as an alternative to MPU-based protection. These results

demonstrated that CHERI can serve as a practical and effective foundation for secure and efficient embedded virtualization systems. The paper was submitted to IEEE Access and is detailed in Chapter 8.

1.2.4 Other Contributions

In addition to the primary research contributions, we participated in several collaborative projects closely aligned with the scope of this thesis. We participated in the Shaheen project [108], which delivers a secure and scalable RV64 SoC targeting nano-Unmanned Aerial Vehicle (UAV) applications, and contributed to other system-level components related to virtualization, such as an early implementation and evaluation of the RISC-V Advanced Interrupt Architecture (AIA) [109], and a complete virtualization-enabled prototype CVA6-based SoC with support for AIA and an Input/Output Memory Management Unit (IOMMU) [110]. We also contributed to the open-source community by upstreaming the hypervisor extension to the mainline CVA6 repository, and we are actively engaged in the process of submitting CHERI support to CVA6 to create a sustainable and reproducible platform for future research on memory safety and isolation in RISC-V systems.

1.3 Thesis Outline

This thesis is organized into two main parts. The first part, comprising Chapters 2 to 3, introduces the foundational concepts and reviews the state of the art. Chapter 2 presents the necessary background, including key features of the RISC-V ISA, the CHERI protection model, virtualization and hypervisor architectures, and tagged memory systems. Chapter 3 presents a survey of related work on mechanisms to enforce memory safety, covering both hardening and mitigation approaches. In addition, it examines compartmentalization and isolation technologies in the context of embedded real-time and MCSs. Furthermore, we incorporated a discussion that positions CHERI within the landscape of existing technologies, complemented by a gap analysis. Finally, Chapter 4 concludes the thesis by outlining future research directions and potential extensions of this work.

The second part contains the main scientific contributions of the thesis and is structured into Chapters 5, 6, 7, and 8. These chapters correspond to the primary contributions of this thesis and are presented in full or adapted form of the peer-reviewed publications. Chapter 5 presents a comprehensive systematization of the CHERI ecosystem. Chapter 6 describes the design and implementation of a virtualization-enabled platform based on the CVA6 core, including microarchitectural optimizations targeting critical virtualization components such as the MMU and timer subsystem. Chapter 7 presents the integration of CHERI support into CVA6, detailing architectural extensions, microarchitectural enforcement logic, and extensive performance evaluation. Finally, Chapter 8 describes the integration of CHERI into the Bao hypervisor, the protection models enabled by CHERI technology, and quantifies the engineering effort through systematic evaluation based on four metrics (code size and memory footprint, boot time, interrupt latency, and performance overhead).

2

Background

This chapter introduces the key technologies necessary to understand this thesis. It begins by describing the main virtualization techniques, hypervisor taxonomy, and the role of virtualization in mixed-criticality environments. It then traces the evolution of capability-based systems and contrasts them with other access-control models. Next, it describes how CHERI re-imagines these concepts to provide hardware-enforced memory safety and software compartmentalization, essential for mitigating modern attack surfaces. Then, it presents RISC-V, an open and extensible standard ISA that has gained traction in domains that demand flexibility and openness, and summarizes relevant parts of the unprivileged and privileged specifications, including support for virtualization and CHERI through emerging standard extensions. Finally, it introduces CVA6, an open-source, in-order RISC-V processor that serves as the experimental foundation for this research.

2.1 Virtualization

Virtualization is a foundational technology that allows multiple OSs to execute concurrently on a single hardware platform, enabling more efficient utilization of resources. This is achieved through a new software layer, also known as the hypervisor, which abstracts and manages the underlying hardware for the set of hosted VMs. In 1974, Popek et al. [114] formally defined the requirements for classical virtualization, stating that a computer architecture is virtualizable if the "set of sensitive instructions is a subset of privileged ones", that is, instructions affecting or exposing the system's privileged state must be trapped when executed outside of it's privileged context. Over the years, virtualization has gained widespread adoption, particularly in cloud computing infrastructures [115–117]. In this context, virtualization facilitates backward compatibility by enabling unmodified legacy software stacks to run alongside modern systems, avoiding the complexity of tightly integrating heterogeneous software components. More recently, virtualization has also gained traction in embedded systems, in particular in safety-critical domains [1–4], such as avionics and automotive, where it enables the consolidation of MCS while meeting strict isolation, safety, and real-time requirements [50, 118, 119] (see Subsection 2.1.3).

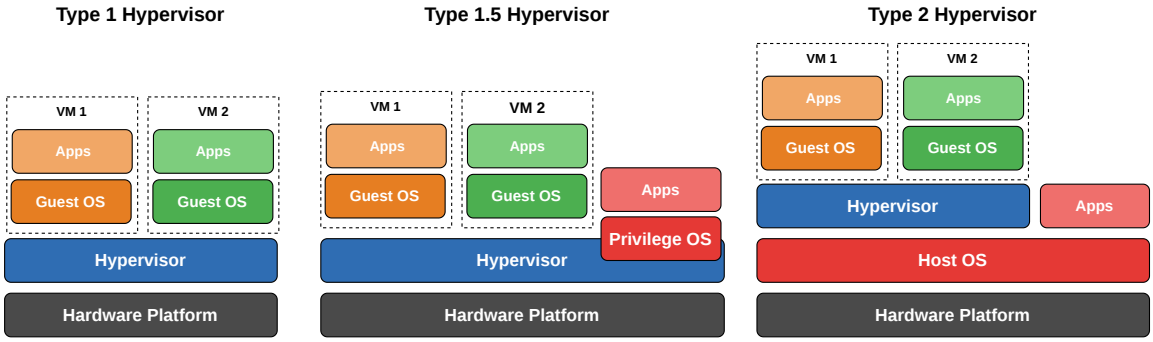


Figure 2: Comparison of Type-1, Type-1.5 (hybrid), and Type-2 hypervisors.

2.1.1 Virtualization Techniques

There are three main virtualization techniques, each with distinct implementations and trade-offs:

- **Binary Translation.** This technique involves the hypervisor intercepting and dynamically rewriting sensitive guest instructions into safe equivalents that preserve the intended functionality while maintaining control over the hardware. A notable example is VMware Workstation [117]. The key advantage of binary translation is that it allows for the execution of unmodified guest OSs. However, it introduces significant performance overhead due to the dynamic rewriting of instructions during runtime.

- **Paravirtualization.** This technique requires modifications to the guest OS. Sensitive instructions are replaced with explicit `hypercalls` to the hypervisor, creating a symbiosis between the guest and the host. Although paravirtualization achieves lower overhead compared to binary translation, it requires changes in the guest OS kernel, making it unsuitable for running proprietary or legacy systems without modification.

- **Hardware-Assisted Virtualization.** This technique leverages dedicated architectural extensions to accelerate virtualization-sensitive operations, such as memory management and trap handling, thereby reducing the reliance on software emulation. Examples of such extensions include the Intel VT-X [54], the AMD-V [120], the Arm VE [57, 121], and the RISC-V hypervisor extension [56]. This approach allows unmodified guest OSs to run with minimal performance overhead.

2.1.2 Hypervisor Taxonomy

Hypervisors are the intermediate software layer that manages and isolates VMs. The literature typically divides these into three types of hypervisors [122], as illustrated in Figure 2.

- **Type-1 Hypervisors.** Execute directly on hardware without relying on a host OS. This type of hypervisors are commonly used in the safety and real-time domains because they typically have a

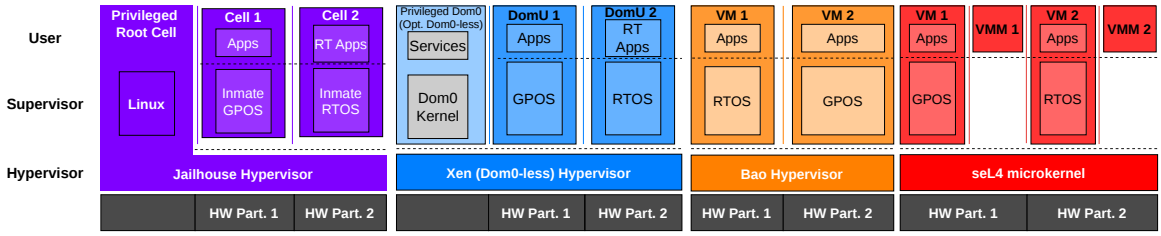


Figure 3: Design space of SPHs for MCS. Adapted from [131].

small TCB and provide stronger real-time guarantees. Examples include: Bao [118], ACRN [123], PikeOS [124], Xen [115, 125], QNX Hypervisor for Safety [126], Xtratum [15], and Wind River Hypervisor [127].

- **Type-2 Hypervisors.** These hypervisors run as user-space applications on top of a general-purpose host OS, taking advantage of the host OS to implement core system functions such as CPU scheduling, memory management, and I/O operations. This model typically incurs higher overhead, increased attack surface, and larger TCBs. Representative examples include: QEMU [128], VMware Workstation [117], and VirtualBox [129].

- **Type-1.5 (Hybrid).** Type-1.5 hypervisors, also referred to as *hybrid hypervisors*, blend characteristics of both Type-1 and Type-2 designs. They are typically implemented as kernel modules within a general-purpose host OS, allowing them to take advantage of hardware virtualization extensions and existing kernel subsystems such as device drivers and schedulers. This architecture avoids the complexity and maintenance burden of developing a standalone Type-1 hypervisor from scratch by reusing the host OS infrastructure. Representative examples include KVM [122], which integrates virtualization support into the Linux kernel, and eMCOS Hypervisor [130].

2.1.3 Virtualization in Mixed-Criticality Systems

In recent years, embedded systems stacks have become highly complex due to the need to consolidate several software stacks of different levels of criticality onto the same hardware platform, also called MCS. This trend is driven by stringent market constraints to optimize SWaP-C, particularly in domains such as automotive, avionics, industrial automation, and medical devices [1–4]. Virtualization has been the key enabling technology for efficiently consolidating MCSs while guaranteeing isolation [50]. A central requirement in such systems is *freedom from interference*, which mandates that the execution of critical tasks must not be adversely affected by the behavior of non-critical components [118, 131]. Hypervisors must provide strict spatial and temporal isolation to prevent data leakage and resource contention between VMs. To address these requirements, hypervisors often enforce isolation by statically partitioning resources at boot time, avoiding dynamic resource sharing, and ensuring deterministic execution at runtime. Hypervisors following this approach are commonly referred to as SPHs [131].

Design Space of Static Partitioning Hypervisors for Mixed-Criticality Systems. SPHs share a common goal of strong isolation and predictability, but differ significantly in their architectural organization and resource management models, as depicted in Figure 3. Jailhouse [132] relies on a Linux-based `root cell` for system initialization, after which the hypervisor takes control and partitions the system into `cells` or isolated domains. In contrast, Bao [118] is a fully standalone hypervisor that boots directly on the hardware without any dependency on a privileged OS. Both of these systems implement pure static partitioning, with a one-to-one mapping of virtual CPUs to physical CPUs, support direct device passthrough, and employ no scheduling, enabling low overhead and high determinism. Xen [115] originates from a feature-rich virtualization architecture where a privileged virtual machine, referred to as Dom0, is responsible for system management, device drivers, and the creation and control of non-privileged VMs or `DomUs`. In this traditional model, Dom0 runs a full operating system (typically Linux) and plays a central role in resource management and I/O virtualization. To address the requirements of embedded and automotive systems, Xen has been extended with a Dom0-less mode [125], where the hypervisor boots guest VMs directly without relying on a privileged OS to initialize the system. In contrast, seL4 adopts a microkernel-based architecture, where hypervisor functionality is decomposed between a small, formally verified kernel and user-level Virtual Machine Monitors (VMMs), typically one per VM, as depicted in Figure 3. When operating as a hypervisor, seL4 executes in a privileged virtualization mode (e.g., HS-mode in RISC-V) and exposes virtualization-specific abstractions through its capability system and Application Programming Interfaces (APIs). In this model, VMMs run as user-space components that leverage these capabilities to create, configure, and manage virtual machines, with each VMM responsible for a single VM. Resources are statically assigned and controlled through capabilities, ensuring strict access control and strong isolation guarantees. This design enables a high degree of assurance, supported by formal verification of the kernel and its security properties. However, the separation between the microkernel and user-level VMMs introduces additional overhead due to increased context switching and interaction between components, which impact performance compared to monolithic SPHs [131].

2.1.3.1 Bao

Bao is a lightweight Type-1 SPH designed specifically for embedded multicore systems with heterogeneous mixed-criticality workloads. Its main goal is to address the needs of safety-critical domains by providing strong spatial and temporal isolation, minimal interference, deterministic execution, and a reduced TCB. Bao exclusively assigns resources to VMs at boot time (e.g., CPUs, memory regions, interrupts, and devices), avoiding any dynamic scheduling of VMs. Bao has been instrumental in the advancement of virtualization support for emerging architectures. In particular, it has been widely adopted within the RISC-V ecosystem as a reference system for validating the hypervisor extension, making it one of the earliest hypervisors to support this extension.

Interference Mitigation. Bao incorporates several mechanisms to mitigate interference arising from the sharing of microarchitectural resources [17]. In particular, it supports cache partitioning techniques

such as cache coloring to reduce contention in the Last-Level Cache (LLC), enabling stronger isolation between VMs at the microarchitectural level. Additionally, Bao allows direct injection of interrupts into guest domains, avoiding costly hypervisor-mediated trap-and-emulate paths and significantly reducing interrupt latency and jitter. Beyond these core mechanisms, Bao has also been used as a research platform to explore advanced isolation and resource management techniques targeting inter-VM interference. For example, works such as MemGuard [133] and H-MBR [134] introduced memory bandwidth regulation mechanisms to reduce inter-VM interference by enforcing bandwidth budgets and per-VM regulation quota. Similarly, Interrupt Request (IRQ) coloring techniques [135] extended the concept of resource partitioning to the interrupt subsystem, mitigating the interference caused by interrupt handling between domains.

Device Sharing. Bao has also been extended to support safe and efficient device sharing in real-time systems. BiRtIO [136] introduces a real-time-aware implementation of VirtIO [137], enabling the deterministic sharing of network interfaces between VMs while preserving isolation. VirtIO is a widely adopted standard for paravirtualized device sharing, particularly in virtualization platforms targeting automotive and embedded domains due to its flexibility and ecosystem support. By incorporating a real-time-aware VirtIO backend, Bao is closer to an industry-ready solution, bridging the gap between research-oriented hypervisors and practical deployments in MCS.

Memory Isolation Models and Architectural Exploration. Bao provides a flexible memory isolation model that supports both MMU-based isolation, requiring hardware support for a second stage of address translation, and MMU-less isolation [138], where memory protection is enforced using a double-stage MPU (e.g., Supervisor Physical Memory Protection (SPMP) for the hypervisor in RISC-V or a dual-stage MPU on Armv8-R platforms [138]). Furthermore, Bao has been used to explore new architectural paradigms. For example, Bao-Enclave [139] leveraged virtualization technology to provide isolated enclave-like execution environments in the normal world, allowing relocation of security-sensitive functionality from the TEE while reducing the overall system TCB.

2.1.3.2 MMU-less Virtualization

Traditionally, MMU-based protection introduces non-deterministic latencies caused by microarchitectural components, such as TLB misses and page table walks, which can jeopardize execution timing guarantees. This is especially critical in automotive Electronic Control Units (ECUs), where safety standards (ISO 26262 [21]) require strict *freedom from interference* for real-time control functions. In contrast, MMU-less architectures, typically based on hardware MPUs, provide a lightweight and more predictable alternative by enforcing memory isolation through region-based protection without address translation. This approach is commonly adopted in Microcontroller Unit (MCU)-class ISAs such as Armv8-R, where PMSAv8 MPU replaces virtual memory, and in RISC-V platforms, where emerging extensions such as dual-stage SPMP for hypervisor enable virtualization without relying on page tables [138]. Similar MPU-based virtualization support is also found in automotive-oriented architectures such as Renesas RH850 and Infineon AURIX TC4Dx. Furthermore, MMU-less isolation mechanisms align with security guidelines such as ISO/SAE

21434 [140], by reducing the attack surface and improving control over memory access, thus enhancing the system resilience to software-based threats [141].

2.2 Capability Systems

2.2.1 What is a Capability?

In 1966, Dennis and Van Horn [66] first defined a capability as an "unforgeable token of authority" used to control access to system resources in multiprogrammed environments. A capability encodes both a reference to an object (e.g., memory, files, or devices) and the rights associated with that object (e.g., read, write, and execute). In this model, capabilities are first-class entities that can be held, passed, restricted, or delegated by programs, but they cannot be forged or fabricated by unprivileged code. This ensures that all authority within the system flows through explicit delegation, rather than implicit identity or privilege. The original idea revolved around enforcing access protection using per-process capability lists, or *C-lists*. Each process maintains a private table of capabilities, and all operations on protected resources must be performed through entries in this list. Capabilities held in *C-lists* are validated and unforgeable. As a result, they enable fine-grained, decentralized access control, supporting confinement, least privilege, and secure composition by design [69, 142].

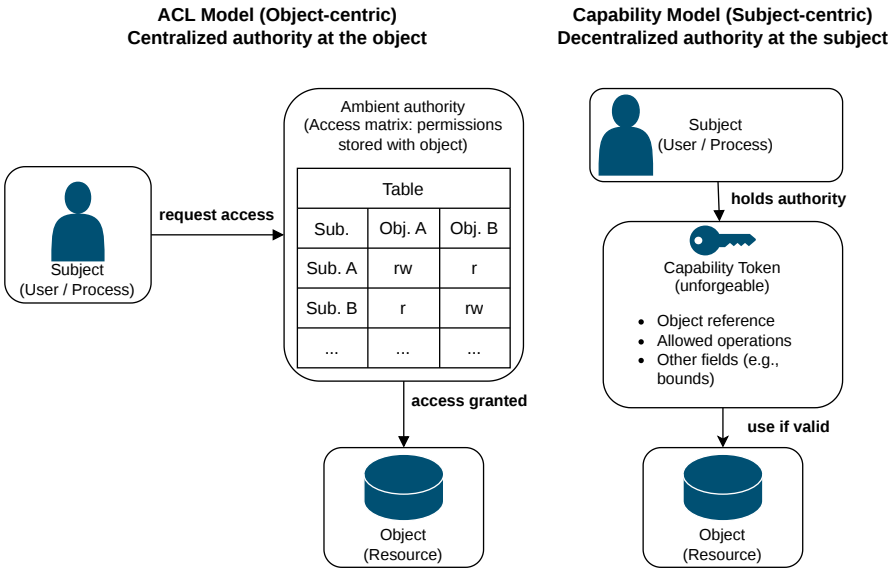


Figure 4: Conceptual comparison between ACL-based and capability-based access-control models. ACLs enforce object-centric protection through centralized permission lists, while capabilities enable subject-centric control by associating authority with unforgeable tokens held by subjects.

2.2.2 Capabilities vs. Access Control Lists

Capabilities and ACLs represent two fundamentally different mechanisms for enforcing access control, as illustrated in Figure 4. ACLs store a list of permissions directly associated with an object, specifying which users or roles can perform certain operations. In contrast, capabilities bind permissions to the subject via unforgeable tokens that reference a specific object and encode the allowed operations. This represents a conceptual shift from object-centric protection, where access is governed by centralized policies at the object level, to subject-centric control, where authority is defined by the possession of specific tokens.

This distinction was formalized by Butler Lampson in his Access Matrix model [70], which showed how the same authorization state can be represented from either an ACL-based or a capability-based perspective. Capabilities, by enabling explicit and decentralized authority flow, eliminate reliance on global identity and reduce the risk of unintended privilege escalation. In contrast, ACLs can introduce ambient authority, where access is granted implicitly based on identity or system context. This vulnerability was famously demonstrated in Hardy's "Confused Deputy" attack [71], which showed how ambient authority systems can allow untrusted code to misuse higher-privilege components to escalate its authority, a flaw that capability systems are specifically designed to prevent.

2.2.3 Evolution of Capability Systems

Early computer architectures enforced memory protection and isolation using segmented memory, a hardware mechanism that partitioned the address space into distinct segments, each defined by a base, size, and access rights. This approach, seen in systems like Multics [143] and Burroughs B5000 [144], provided one of the first forms of memory segmentation. Although not full capability systems, segment descriptors resemble early forms of capability enforcement. This model introduced foundational concepts that were later adopted by capability systems, notably, the association of access rights with address references and the support for fine-grained software compartmentalization.

Capability-Based Microkernels. Microkernels are OS kernels designed to minimize the TCB by implementing the bare minimum features in the kernel space, such as scheduling, memory management, and IPC, and moving other services (e.g. file systems, drivers) to user space [145]. Capability-based microkernels use capabilities as the fundamental mechanism for providing security, fine-grained access control, and resource management. Early systems such as Hydra [67] and KeyKOS [72] pioneered this design in the 1970s, demonstrating that capability-based protection enables fine-grained access control. Later systems such as EROS [68] advanced these principles and improved performance and simplified the revocation mechanism. Today, microkernels such as Fiasco [52], seL4 [51], and Google's Fuchsia [146] combine capability-based design with modern engineering goals. The seL4 [51] microkernel formally verifies the capability model with machine-checked proofs of correctness, offering mathematical guarantees of both safety and security properties at the implementation level. Fiasco [52], from the L4 family, emphasizes high performance and real-time capability-based resource management. Fuchsia [146] adopts

a similar principle of fine-grained capability control for a secure modular system. Together, these systems demonstrate how the foundational ideas of Hydra [67], KeyKOS [72] and EROS [68] influenced the development of practical, verifiable, and industry-scale microkernels.

Hardware Capability Architectures. Early hardware capability systems explored embedding fine-grained authority and object protection directly into the processor and memory hierarchy. Among industrial efforts, the Plessey System 250 [147, 148] was a pioneering example that implemented tagged memory and capability-based addressing to enforce isolation and controlled delegation on the hardware. The Intel iAPX 432 [149, 150] represented a commercial attempt to bring capability and object-based protection into a general-purpose processor, integrating hardware objects, descriptors, and bounds enforcement. However, it suffered from high complexity, limited performance, and poor software compatibility, leading to its eventual discontinuation. In parallel, a series of research-oriented capability machines, including the CAP Computer [151, 152], the Flex [153, 154], M-Machine [77], PICA [155, 156], SAFE [157], and CODOMs [158], explored different designs for capability-based protection. Flex [153, 154] was one of the first to investigate the storage of capabilities using tagged memory. M-Machine [77] explored capability-aware microarchitectures. PICA [155, 156] and SAFE [157] focused on verifiable control-flow and fine-grained compartmentalization protection. CODOMS [158] explored the integration of capability-based protection into conventional hardware protection models. It aimed to retain compatibility with existing software stacks while introducing object-level access control with minimal software changes.

The Downfall and Rise of Capability Systems. Despite their conceptual advantages, capability-based systems failed to achieve commercial adoption by the late 1980s and early 1990s, as the computing landscape shifted toward performance-oriented rather than security-oriented architectures. Several factors contributed to this decline. First, managing capabilities required complex instructions, such as in the Intel iAPX 432 [149, 150], especially when invalidating capabilities across distributed processes (revoking capabilities) [69]. Second, the emergence of Reduced Instruction Set Computer (RISC) architectures, such as Microprocessor without Interlocked Pipeline Stages (MIPS) [159] and Scalable Processor ARChitecture (SPARC) [160], shifted design priorities toward simplicity and instruction throughput rather than complex instructions [79, 80]. Capability-systems introduced additional overhead to designs, often requiring additional hardware metadata and checkers. Third, silicon and memory constraints made it impractical to dedicate on-chip resources to capability tags, bounds registers, and protected lookup tables, especially at a time when transistor budgets were tightly constrained. Finally, conceptual misconceptions [161] and ecosystem inertia hindered adoption because capabilities were often misunderstood as equivalent to ACLs, leading many designers to favor more conventional mechanisms such as ring-based privilege levels, MMU-based paging, and identity-based security models aligned with UNIX-based systems. Interest in capability-based protection reemerged in the 2000s, driven by growing system complexity, expanding attack surfaces, widespread memory safety vulnerabilities [24, 25], and the increasing prevalence of interconnected and safety-critical devices. This resurgence was reflected in systems like Capsicum [162], which introduced capability semantics to UNIX-like environments through file descriptor delegation and

sandboxing. These developments laid the foundation for CHERI [37], which reintroduces the classical capability model into modern RISC architectures while remaining compatible with traditional protection mechanisms (e.g., ring levels and paging).

2.3 CHERI

The CHERI architecture extends conventional ISAs with a hardware-accelerated capability protection model that enforces fine-grained memory protection and scalable software compartmentalization. At a high level, CHERI replaces traditional integer pointers with architectural capabilities that are unforgeable references that convey an address, bounds, and the rights to access it. This enables the enforcement of spatial memory safety and fine-grained access control directly in hardware.

CHERI Core Design Principles. The protection model of CHERI is based on the following core security design principles. First, the *Principle of Least Privilege*, which states that each actor in a system must have the minimum access rights necessary to perform its tasks, preventing the potential impact of malicious exploits [143]. Second, the *Principle of Intentional Use*, which states that "whenever multiple rights are available to a program, the selection of rights used to authorize work on behalf of the program must be explicit" [163].

2.3.1 CHERI Protection Model

At the core of the CHERI architecture lie the ISA-defined architectural capabilities that replace and augment traditional pointers with capability-based protection. A CHERI capability is a 128-bit word that encodes a base address, memory limits (length), permissions, and a validity tag. The tag bit is an out-of-band bit, inaccessible to software, which ensures the *capability integrity*. In this model, all memory accesses are mediated by capabilities, and operations such as loads, stores, and control transfers are extended to operate on capability registers. This design enforces spatial memory safety at the hardware level, preventing classical memory safety issues, such as buffer overflow. The CHERI capability extension defines semantics to ensure that: (i) capabilities can only be derived from valid capabilities with equal or lesser authority, ensuring no privilege escalation (`valid provenance` and `monotonicity` properties) and (ii) the tag bit prevents software from forging capabilities, ensuring that invalid operations clear the tag and invalidate the capability (`integrity` property). In addition, CHERI provides an object-type field that can be leveraged to implement object encapsulation, secure return-oriented control transfers, and enforcement of abstract data types at the hardware level.

Software Compartmentalization and Object Sealing. Beyond the spatial memory protection offered by the capabilities, the CHERI capabilities can also be used to provide secure, fine-grained, and scalable software compartmentalization. Compartments are pieces of software that run in isolation from

other modules and have well-defined boundaries and interfaces. CHERI provides a new type of capability, `sealed capabilities`, which in practice are opaque object references that must be explicitly unsealed before use. Sealing prevents unauthorized inspection or modification of encapsulated objects, forming the foundation for strong fault isolation and mutual distrust.

2.3.2 CHERI and Legacy Protection Mechanisms

CHERI supports multiple design configurations that blend legacy and capability-based protection mechanisms [37, 82]. As illustrated in Figure 5, these configurations define a spectrum of system designs ranging from fully legacy systems to fully pure-capability systems, enabling incremental adoption of CHERI while maintaining compatibility with existing software stacks.

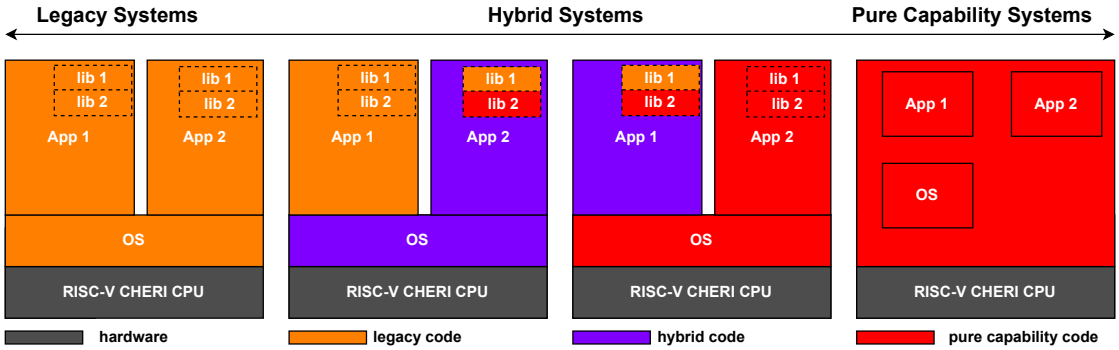


Figure 5: Architectural execution modes and deployment configurations supported by CHERI, ranging from hybrid systems combining legacy and capability-based protection mechanisms to fully pure-capability architectures. Adapted from [82].

Legacy Systems. In legacy systems, protection relies entirely on conventional mechanisms such as the MMU and ring-based privilege levels. The MMU enforces isolation through virtual memory and page-level permissions, while privilege levels regulate access to sensitive operations (e.g., user and kernel modes). In this configuration, pointers are unprotected and can be freely manipulated, leaving the system vulnerable to memory safety violations. CHERI is not used in this model, and all isolation is coarse-grained and is externally enforced by the hardware and OS.

Hybrid Systems. Hybrid systems combine legacy protection mechanisms with CHERI capabilities. In this configuration, systems retain the MMU and privilege levels for coarse-grained isolation across processes and privilege domains, while CHERI introduces fine-grained protection within an address space. Applications may consist of a mix of legacy code (using integer pointers) and capability-aware code, where capabilities must be explicitly introduced through compiler annotations. This model enables incremental adoption, allowing unmodified binaries to coexist with capability-enhanced components. Here, CHERI complements traditional mechanisms by enforcing spatial memory safety and enabling intra-process compartmentalization, while the MMU and privilege levels remain responsible for process-level isolation and system security boundaries.

Pure-Capability Systems. At the opposite end of the spectrum, pure-capability systems enforce protection exclusively through CHERI capabilities. In this configuration, all pointers are represented as capabilities, and memory access is governed by capability bounds and permissions. In such systems, traditional privilege-level separation and virtual address space isolation are not strictly required to separate software components, since an OS and applications can, in principle, execute at the same privilege level while authority is constrained by the capabilities they hold. This approach highlights CHERI's ability to serve not only as a complementary protection mechanism, but also as a foundational substrate for fine-grained system isolation.

Execution Modes. To support this range of configurations, CHERI defines two main modes of execution. In `integer mode`, programs use a combination of capabilities and legacy pointers, enabling hybrid operation and gradual adoption. In contrast, `the pure-capability mode (purecap)` completely eliminates legacy pointers, enforcing that all memory accesses are performed through capabilities. This mode provides the strongest security and memory safety guarantees.

2.3.3 Architectural Support: RISC-V and Arm Morello

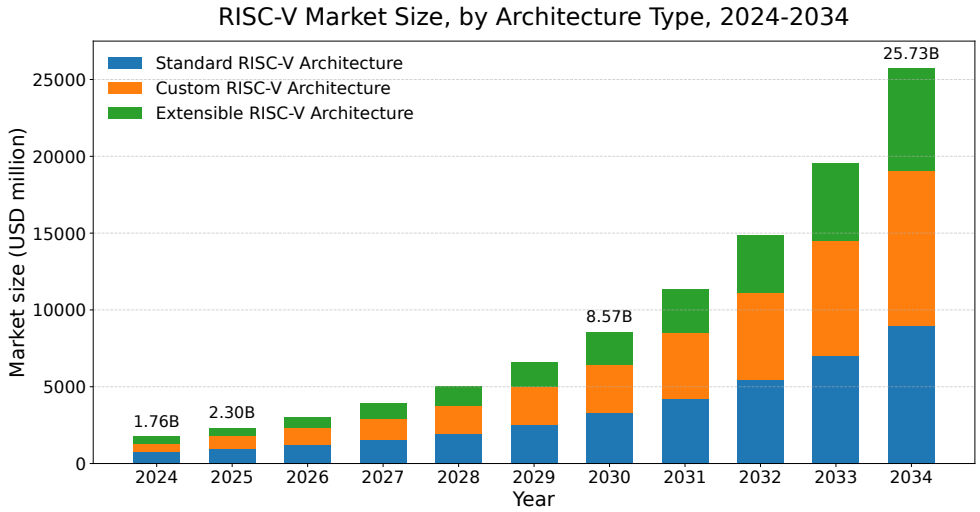
CHERI has been implemented in multiple ISAs. In RISC-V, several research and industrial projects have emerged around CHERI [164]. Notable examples include the first prototypes from Cambridge (e.g., Flute [165] and Toooba [166]), and the Microsoft initiative, the CHERIoT project [87], to scale the architecture for deeply embedded systems with a lightweight capability model. These efforts demonstrate CHERI's versatility across a wide performance and complexity spectrum, from high-performance out-of-order cores to microcontroller-class devices. The CHERI RISC-V ISA is now at a turning point, as several key players (e.g., Codasip, University of Cambridge and Capabilities Limited) are contributing to the standardization of a set of CHERI extensions (*Zcheri* [107]), which paves the way for future silicon implementations and wider adoption. In the Arm ecosystem, CHERI is realized in the *Arm Morello* prototype platform [83, 167], a research extension to the Neoverse N1 architecture. *Arm Morello* provides a full 64-bit pure-capability execution environment, allowing the evaluation of CHERI principles at scale and serving as a roadmap for future architectural adoption.

2.4 RISC-V

RISC-V [168] is an open-standard ISA designed for modularity, extensibility, and specialization. Implementations start from a compact base integer ISA and comprise optional standard extensions to match system requirements, scaling from tiny MCUs to application-class processors capable of running a full OS, e.g., Linux. This composability enables tight hardware–software co-design while avoiding typical licensing constraints of proprietary ISAs (e.g., Arm).

2.4.1 RISC-V Market Growth

Analyst forecasts indicate a rapid adoption of RISC-V in the coming years, with the market expected to reach $8.57B by 2030 and shipments approaching 17 billion processors in 2030, corresponding to a Compound Annual Growth Rate (CAGR) of approximately 30.7% over 2025-2034 [169–171] (see Figure 6). This acceleration is driven by three key forces:



Source: Global Market Insights, RISC-V Market Size & Share Report, 2025-2034. Intermediate yearly values and segment splits are estimated from published anchors.

Figure 6: RISC-V market size by architecture type from 2024 to 2034, showing projected growth across standard, custom, and extensible architectures. Adapted from [169].

- **Silicon freedom.** RISC-V is a royalty-free open standard ISA, offering clear advantages over proprietary alternatives by eliminating licensing costs and reducing vendor lock-in [169].
- **Heterogeneous computing.** Closed architectures prevent customization for specific workloads. The rise of Artificial Intelligence (AI) and IoT increases the need for architectural flexibility, which RISC-V provides through its modular and extensible ISA, enabling domain-specific extensions and tightly integrated heterogeneous SoCs [172]. This is especially valuable for AI acceleration (e.g., Neural Processing Units (NPUs)) and emerging paradigms such as neuromorphic computing [173]. In parallel, the automotive sector is shifting to zonal/domain controllers with on-board AI (e.g., ADAS, perception, infotainment) that benefit from customizable compute [169].
- **Semiconductor sovereignty.** Geopolitical dynamics are driving policies aimed at semiconductor sovereignty and open architectures, for example, China's national directives to boost RISC-V adoption [174] and the EU Chips Act [175] to strengthen the semiconductor ecosystem.

2.4.2 RISC-V ISA Overview

The RISC-V ISA is organized into the (i) unprivileged [176] and (ii) privileged specifications [177].

RISC-V Unprivileged Specification. The Unprivileged Specification [176] defines the base integer ISAs that every RISC-V processor builds upon, the `RV32I` and `RV64I`, and their embedded variants with a reduced register file (`RV32E`/`RV64E`). It specifies the general-purpose registers, instruction encodings, and the minimal instruction set required to run RISC-V programs: arithmetic/logic, load/store, and control-flow (branches, `JAL`/`JALR`), as well as user-level system operations such as `ecall`/`ebreak` and memory fences (`fence`, `fence.i`). It also defines the memory model: RISC-V adopts a weak ordering model (`RVWMO`) that favors hardware flexibility and performance. Stronger ordering is established explicitly via memory fences or by using the atomic instructions. In addition, the Unprivileged Specification standardizes widely used extensions, including (i) integer multiply/divide (`M`), (ii) atomics (`A`), (iii) compressed 16-bit encoding (`C`), (iv) single- and double-precision floating point (`F`/`D`), (v) the scalable vector extension (`V`), (vi) control/status register access instructions (`Zicsr`), and (vii) instruction synchronization instructions (`Zifencei`).

RISC-V Privileged Specification. The RISC-V privileged architecture [177] provides the mechanisms required to execute full software stacks, including firmware, OSs, and hypervisors, while enforcing isolation, regulating privilege transitions, and ensuring efficient trap handling. Table 3 summarizes the main components of the specification, organized according to their role in system execution, from privilege management and memory translation to interrupt handling and platform support. We highlight in blue the components related to the hypervisor extension, which are further explored in Chapter 6.

At its core, the privileged specification defines a hierarchical privilege model composed of machine (M), supervisor (S), and user (U) modes. This hierarchy enables a clear separation of responsibilities: M-mode executes firmware, S-mode runs the OS, and U-mode executes regular applications. The architecture provides fine-grained control over privilege transitions and trap delegation, allowing higher privilege levels to selectively delegate interrupts and exceptions to lower levels, reducing unnecessary transitions to firmware. Memory management is supported through both virtual memory and physical protection mechanisms. When an MMU is present, multiple paging schemes (e.g., `Sv32` and `Sv39`) enable virtual address translation and process isolation. In parallel, Physical Memory Protection (PMP) provides a lightweight mechanism to enforce physical memory protection, particularly relevant for systems without virtual memory or to protect critical regions from software running in M-mode (e.g., firmware).

Beyond core execution support, the standardized privileged architecture includes system-level components such as Hardware Performance Monitor (HPM), which exposes execution metrics and timer facilities (e.g., Core-Local Interrupt Controller (CLINT)). These components are essential for OS services, scheduling, and performance analysis. Finally, the privileged specification is complemented by platform-level standards, such as AIA for advanced interrupt delivery and IOMMU for I/O device isolation, as well as Supervisor Binary Interface (SBI), which standardizes the interface between firmware and OS.

Component	Description
	Privilege Levels
Machine Mode (M-mode)	Highest privilege, runs the firmware.
Supervisor Mode (S-mode)	Runs the OS.
User Mode (U-mode)	Runs the applications.
	Virtual Memory
Paging Support	Translates virtual addresses to physical addresses, implemented by the MMU and controlled by the OS. Supports multiple translation schemes: `bare` (no translation), `Sv32` (32-bit address spaces), `Sv39` (39-bit address space), `Sv48` (48-bit address space), `Sv57` (57-bit address space).
	Trap Handling
Interrupts/Exception Handling	Interrupts are categorized as software, timer, and external. Each privilege level has its own set traps, with interrupts exposed through the `xip` CSRs and enabled through the `xie` CSRs. Highest privileged modes can delegate interrupts/exceptions to lower privilege modes (via `xideleg` CSRs) to control trap behavior.
Exception Handling	There are also multiple sources of exceptions, including page faults, illegal instruction, etc. Highest privileged modes can delegate exceptions to lower privilege modes via `xedeleg` CSRs. The `xcause` register provides the exception encoding upon trap handling.
	Memory Protection
PMP	Managed by M-mode, protects critical physical memory regions from lower privilege levels using a small set of configurable entries with access permission bits (R, W, X), address matching modes (`NAPOT`, `TOR`).
	Performance Monitors
HPM	Up to 29 programmable counters (e.g., instructions retired, CPU cycles) via `mhpmcounter` and `mhpmevent`. Available in M-mode, optionally exposed to S/U.
	Timers & IPIs (CLINT)
CLINT	Provides timer and software interrupts. Contains `mtime`, `mtimecmp`, `msip` for inter-processor interrupts and time management.
	Hypervisor Extension
Privilege Levels	Introduces `HS`, `VS`, `VU` modes. HS-mode is S-mode extended with hypervisor funcionalities to run a hypervisor and manage guest execution. VS and VU run the guest OS and application, have a their own set of shadow S-mode CSRs.
Interrupts/Exceptions	Supports direct-interrupt injection via `vgein`, `VSIE`, `VGEIP`. Needs platform level support. Exceptions are virtualized using trap redirection. Supports virtual interrupt injection.
Two-stage translation	Supports a new stage of translation (G-stage) controlled by the hypervisor that translates guest physical to host physical and the VS-stage (or S-stage) controlled by the VM that translates guest virtual to guest physical. Managed by `hgatp`, `vsatp`.
Hypervisor Instructions	Provide instructions for peeking at the guest state and control the translation caches.
	Platform-Level Standards
AIA	The AIA defines the RISC-V standard interrupt controllers that support: Message Signaled Interrupts (MSI), Inter-Process Interrupts (IPIs), direct-interrupt injection to the guest, delegation of interrupts to lower privilege contexts.
IOMMU	Standard specification that adds address translation and protection for I/O devices with DMA-capabilities. Also supports virtualization by adding a second stage of translation.
Input/Output Physical Memory Protection (IOPMP)	The IOPMP defines the standard physical memory protection mechanism for DMA-capable I/O devices. Protects physical memory regions from unwanted I/O-device memory accesses.
	SBI Interface
SBI	Standardizes the firmware-OS interface. Provides services: timer, IPI, shutdown, remote fence, hart start/stop. OpenSBI is the most popular RISC-V firmware.

Table 3: Summary of RISC-V privilege specification

2.4.3 CVA6

CVA6 is an open-source application-class RISC-V core maintained by the OpenHW Group as part of the CORE-V program [178]. The design is highly configurable through build-time parameters, enabling adaptation to a broad spectrum of workloads, ranging from resource-constrained embedded systems without an MMU to fully fledged Linux-class platforms. The core can be configured for either `RV32` or `RV64`, supports the standard RISC-V privilege levels (M/S/U), and includes virtual memory support. Optionally, it also supports the RISC-V hypervisor extension, which is available in the mainline design and whose implementation is one of the main contributions of this work (see Chapter 6).

Extensibility and ISA Support. CVA6 provides a flexible framework for extending the base RISC-V ISA through the CORE-V eXtension Interface (`CORE-V X-Interface`), which enables the integration of custom coprocessors and specialized functional units. This interface allows tightly coupled accelerators to interact directly with the pipeline, facilitating domain-specific optimizations without requiring intrusive modifications to the core microarchitecture. The core supports a wide range of standard RISC-V extensions, including integer, floating-point, and atomic operations. It has also been used as a platform for integrating advanced architectural features, such as the Ara vector coprocessor [179], which enables efficient execution of vectorized workloads for high-performance AI/Machine Learning (ML) applications. More recent work has further explored the extensibility of CVA6 by integrating architectural support for Control-Flow Integrity (CFI) through the standard RISC-V extensions `Zicfiss` and `Zicfilp` [180].

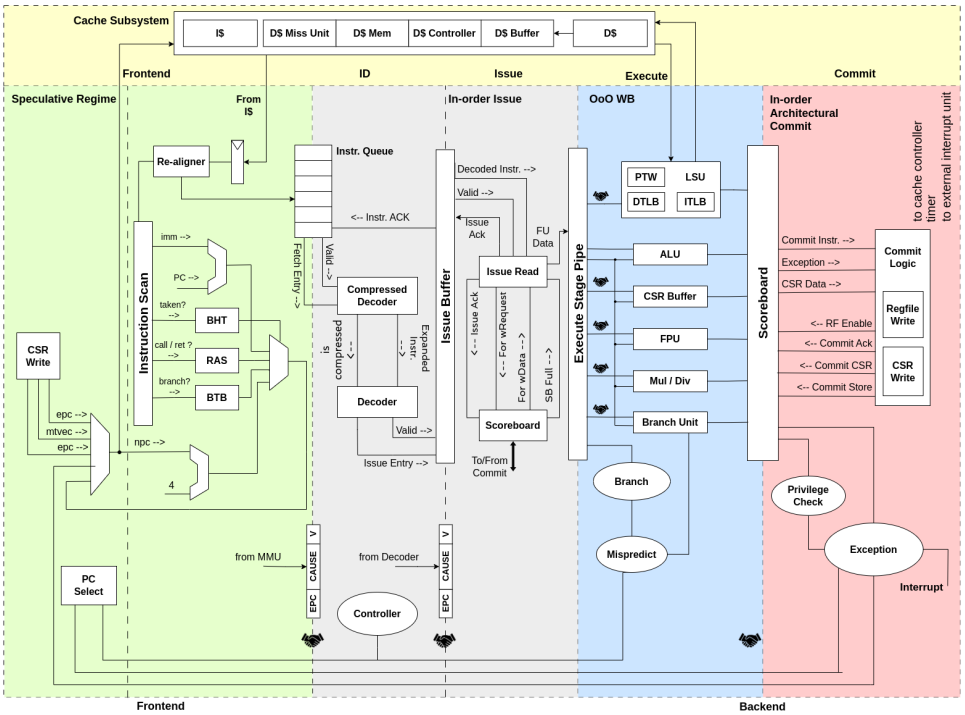


Figure 7: CVA6 core microarchitecture overview. Taken from [178].

Pipeline Microarchitecture. CVA6 implements a single-issue, 6-stage in-order pipeline organized into a frontend and a backend, as illustrated in Figure 7, which are decoupled through a First-In, First-Out (FIFO) instruction buffer to tolerate short-term imbalances between instruction fetch and execution. The frontend encompasses the program counter generation, instruction fetch, and decode stages. It integrates branch prediction structures, including a Branch Target Buffer (BTB) and a Branch History Table (BHT), to enable speculative execution and reduce control hazards. The program counter generation stage determines the next instruction address based on several control signals (e.g., predictions and control-flow events), while the instruction fetch stage retrieves and aligns instructions from the instruction cache. The decode stage translates instructions into internal control signals and identifies operand dependencies. The backend includes the issue, execute, and commit stages, which are responsible for instruction scheduling, execution, and architectural state updates. Instruction dispatch is governed by operand availability and dependency tracking mechanisms, enabling efficient utilization of execution units while preserving in-order retirement. The execute stage integrates the core functional units, including load-store, control-flow, memory, and system operations. Although variable-latency operations, particularly memory accesses, may introduce limited reordering in execution, the architectural state is always updated in-order. The commit stage finalizes instruction execution by updating the architectural state, resolving write-back conflicts, and ensuring precise exception handling. Recent work has extended CVA6 towards superscalar designs [181], introducing multiple-issue pipelines and more complex scheduling mechanisms to improve performance, at the cost of increased microarchitectural complexity.

Cache Subsystem. The cache subsystem in CVA6 is configurable, supporting different design trade-offs depending on the target application. The baseline design employs a set-associative cache organization and can be configured with either write-back or write-through policies, enabling flexibility between performance and simplicity. For multicore deployments, CVA6 can be combined with cache-coherent interconnects such as Culsans [182], providing a scalable coherence protocol to maintain consistency between cores and shared memory hierarchies. Beyond the baseline design, CVA6 also supports more advanced cache implementations. In particular, the HPDCache [183], which is an open-source high-performance L1 data cache designed as a drop-in replacement for the default CVA6 data cache. This cache improves memory access efficiency by providing higher bandwidth, reduced latency, and support for advanced features such as cache management operations.

Platforms and Ecosystem. CVA6 is integrated into a rich ecosystem of open-source platforms and SoC designs, making it a practical target for both research and industrial deployment. A prominent example is Cheshire [184], a lightweight Linux-capable RISC-V host platform developed by the PULP group at ETH Zurich, designed to coordinate domain-specific accelerators within heterogeneous systems. Building on this, Carfield [185] extends the architecture towards MCS by organizing the platform into domains with different requirements in terms of safety, security, and performance. CVA6 has also been adopted in silicon prototypes and large-scale systems, such as Occamy [186], a 2.5D chiplet-based system with 432 cores where CVA6 acts as a management core, and Basilisk [187], an open-source Linux-capable

chip based on Cheshire. These examples demonstrate the suitability of CVA6 for both embedded and high-performance systems.

The CVA6 ecosystem is strongly rooted in the European open hardware landscape, driven by research groups such as the PULP team and supported by collaborative European projects (e.g., ISOLDE and TRISTAN), which promote the adoption of RISC-V in safety and security-critical domains. The OpenHW Group further bridges academic research and industrial deployment, with companies such as Thales actively contributing to and adopting CVA6 in industrial contexts. This strong ecosystem reinforces CVA6's position as a mature and relevant platform, making it a suitable foundation for this thesis.

3

State of the Art

This chapter presents the state of the art underlying this dissertation, with emphasis on: (i) memory-safety mechanisms and (ii) compartmentalization and isolation technologies. We organize the chapter into two parts. The first part (Section 3.1) motivates memory safety as a persistent class of vulnerabilities in system software and introduces a taxonomy of spatial and temporal errors, along with representative exploitation techniques. It then surveys mitigation, detection, and hardening approaches for memory safety (Subsections 3.1.3 and 3.1.4), covering memory-safe programming languages, compiler- and runtime-based enforcement, static and dynamic analysis, fuzzing, formal methods, and hardware-backed defenses, including capability-based designs such as CHERI. The second part (Section 3.2) examines compartmentalization and isolation mechanisms in the context of MCS, ranging from microkernels and hypervisors to microarchitectural isolation and TEEs. The chapter closes by positioning CHERI within this broader landscape, outlining the remaining gaps that motivate this dissertation (Section 3.3).

3.1 Memory Safety: Relevance, Exploits, and Mitigation Techniques

Memory bugs remain one of the most prevalent classes of security vulnerabilities, particularly in low-level programming languages such as C and C++, which lack built-in memory safety or type enforcement. As noted by [188], the pursuit of memory safety has fueled decades of research and engineering, shaped by an ongoing arms race between exploitation techniques and defensive mitigations. Memory safety bugs, such as out-of-bounds, consistently appear among the top three software vulnerabilities in the MITRE Common Weakness Enumeration (CWE) Top 25 ranking [189]. Although a compelling solution would be to rewrite critical systems in memory-safe languages such as Rust [38], this is not feasible in practice for two primary reasons. First, many complex systems, such as OSs, embedded firmware, and hypervisors, are written in millions of lines of legacy C/C++ code, making a complete rewrite prohibitively expensive and time-consuming, thus companies prefer investing in interoperability between safe languages and C/C++, rather than rewriting the code from scratch [26]. Second, even memory-safe languages often

Type	CVE	Year	Description	CVSS v3
Spatial	CVE-2024-4761	2024	Chrome V8 out-of-bounds write via crafted HTML; remote heap-corruption/code-execution class bug.	8.8
	CVE-2024-5159	2024	Chrome ANGLE heap buffer overflow causing out-of-bounds read via crafted HTML.	8.8
	CVE-2024-5499	2024	Chrome Streams API out-of-bounds write; arbitrary code execution inside the sandbox.	8.8
	CVE-2024-10487	2024	Chrome Dawn out-of-bounds write via crafted HTML.	8.8
	CVE-2025-5419	2025	Chrome V8 out-of-bounds read/write via crafted HTML; heap corruption.	8.8
	CVE-2025-4919	2025	Firefox JavaScript object out-of-bounds read/write by array-index confusion.	8.8
	CVE-2025-30312	2025	Adobe Dimension out-of-bounds write; arbitrary code execution in user context.	7.8
	CVE-2026-5282	2026	Chrome WebCodecs out-of-bounds read via crafted HTML.	8.1
	CVE-2026-5873	2026	Chrome V8 out-of-bounds read/write via crafted HTML; arbitrary code execution.	8.8
	CVE-2026-6314	2026	Chrome GPU out-of-bounds write enabling sandbox escape from a compromised process.	8.3
Temporal	CVE-2024-0752	2024	Firefox updater on macOS use-after-free during a busy update; exploitable crash.	6.5
	CVE-2024-7528	2024	Firefox IndexedDB garbage-collection interaction use-after-free.	8.8
	CVE-2024-9680	2024	Firefox animation timelines use-after-free enabling code execution.	9.8
	CVE-2024-1086	2024	Linux kernel `nf_tables` use-after-free enabling local privilege escalation.	7.8
	CVE-2025-24983	2025	Windows Win32 kernel subsystem use-after-free; local privilege escalation.	7.0
	CVE-2025-8292	2025	Chrome Media Stream use-after-free via crafted HTML; heap corruption.	8.8
	CVE-2025-9478	2025	Chrome ANGLE use-after-free via crafted HTML; heap corruption.	8.8
	CVE-2026-2787	2026	Firefox DOM Window and Location use-after-free; critical browser memory corruption flaw.	9.8
	CVE-2026-6317	2026	Chrome Cast use-after-free via crafted HTML; arbitrary code execution.	8.8
	CVE-2026-6358	2026	Chrome XR use-after-free causing out-of-bounds read via crafted HTML.	8.8

Table 4: Examples of spatial and temporal memory-safety CVEs (2024–2026).

require unsafe primitives for low-level operations such as direct memory manipulation, which are essential operations in low-level systems programming [190]. This section explores the landscape of memory safety enforcement. It begins by categorizing the types of memory safety vulnerabilities. Then, it describes state-of-the-art mitigation strategies from a software and hardware perspective.

3.1.1 Taxonomy of Memory Safety Violations

Memory safety violations fall into two categories: (i) spatial memory violations and (ii) temporal memory violations. Representative real-world examples of both classes, drawn from recent Common Vulnerabilities and Exposures (CVE) entries, are summarized in Table 4, along with their scores according to the Common Vulnerability Scoring System (CVSS).

Spatial Violations. Spatial memory violations occur when a program reads from or writes to memory outside the boundaries of an object. Classic examples of such violations include (i) stack-based or (ii) heap-based buffer overflows, which attackers can leverage to perform control-flow hijacking or data leakage [188]. A representative example of this exploit is the CVE-2024-49775 [191], a heap-based buffer overflow vulnerability found in Siemens AG User Management Component (UMC). An unauthenticated remote attacker could exploit this flaw to execute arbitrary code, leak sensitive data, or compromise industrial infrastructure. The vulnerability has been assessed with a high severity score of CVSS-B 9.3.

Temporal Violations. Temporal violations occur when memory is accessed beyond its valid lifetime, for example through dangling pointers after deallocation. Attackers may exploit such errors by retaining references to freed memory and reusing them to compromise critical system components or leak confidential data [188]. A prime example of such an attack is the CVE−2025−29828 [192], a vulnerability in Microsoft Cryptographic Services, where improper memory release after use could lead to remote code execution. It affects multiple Windows versions and has a CVSS-B score of 8.1.

3.1.2 Taxonomy of Memory Safety Protection Techniques

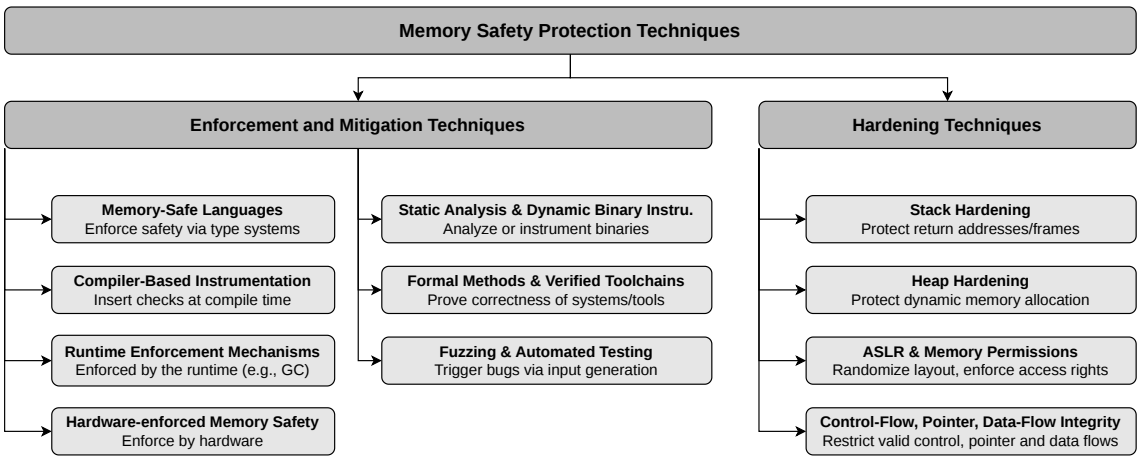


Figure 8: Taxonomy of memory-safety protection techniques considered in this dissertation.

Figure 8 presents the taxonomy of memory-safety protection techniques considered in this dissertation. At a high level, these techniques can be divided into two broad classes. The first class comprises memory-safety enforcement and mitigation mechanisms (Subsection 3.1.3), which aim to prevent, detect, or eliminate memory errors through safer programming models, compiler instrumentation, runtime enforcement, hardware-assisted techniques, formal verification, and automated vulnerability discovery tools. The second class comprises memory-safety hardening mechanisms (Subsection 3.1.4), which do not eliminate the underlying bugs, but instead reduce their exploitability by protecting control-flow state, memory layout, and data integrity. This taxonomy provides a structured view of the design space and guides the organization of the following subsections.

3.1.3 Memory Safety Enforcement and Mitigation Techniques

3.1.3.1 Memory-Safe Languages

A common approach to deal with memory safety, especially in embedded systems, is to adopt a hybrid strategy, focusing on the interoperability of C/C++ with memory-safe languages, such as Rust [38] or Ada [39]. The approach consists of writing new components or rewriting critical components in memory-safe languages, and applying runtime protection or static analysis tools to the C/C++ codebases or unsafe

code blocks (used in Rust to handle low-level operations such as MMIO accesses). In the embedded domain, Rust is a strong candidate to replace C/C++ [193] programming language, especially given that: (i) it has strong interoperability with C/C++, meaning that Rust applications can be built on top of C/C++ codebases, enabling incremental adoption of Rust in the embedded space, and (ii) it achieves performance results comparable to regular C/C++ applications [190]. Several well-known projects have been adopting Rust in their codebases, including the Linux kernel [40], PikeOS [194], and Arm TF-A [195]. Another example of a memory-safe programming language is Ada [39]. It is a high-level statically typed programming language that has been around for more than 40 years and has been widely used to improve software reliability in critical systems. Designed with strong typing and compile-time checks, Ada [39] targets applications where safety, determinism, and certification are of critical importance. Ada is widely used in domains constrained by strict safety standards such as DO-178C [196] (avionics), IEC 61508 (industrial control) [197], and EN 50128 (railways) [198].

3.1.3.2 Compiler-based Instrumentation

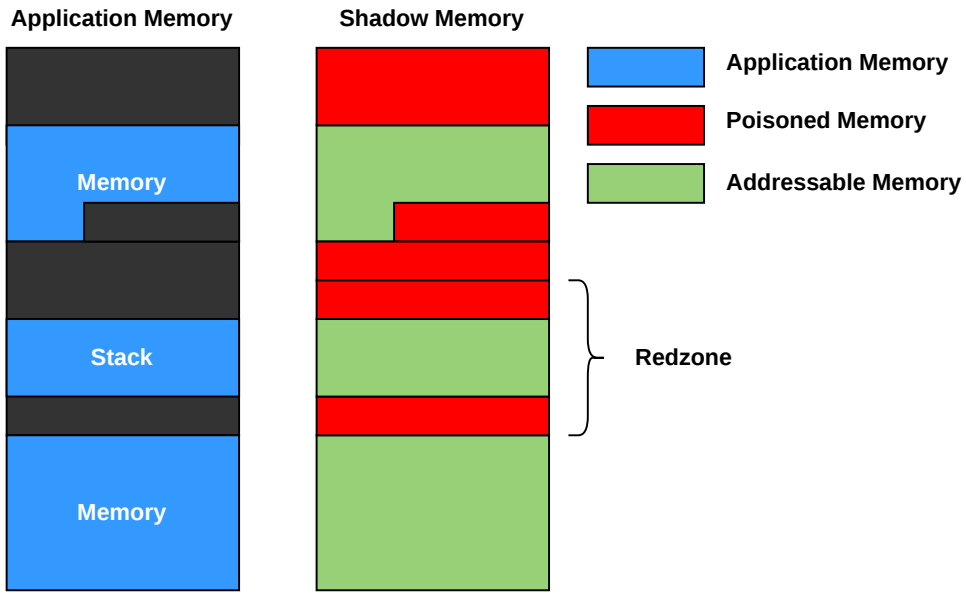


Figure 9: Overview of AddressSanitizer memory layout, where allocated objects are surrounded by redzones that are marked as poisoned in shadow memory. Valid regions remain accessible, while any access to poisoned regions (e.g., redzones or freed memory) is detected at runtime.

Compiler-based instrumentation approaches aim to detect and restrict unsafe memory operations at runtime by inserting additional metadata or instructions into program binaries. Widely used tools in this space include the LLVM/GCC sanitizers. For instance, AddressSanitizer [43] combines compiler instrumentation with a runtime library that uses shadow memory and redzones to detect many spatial memory

errors and common temporal heap errors, such as UAF bugs (see Figure 9). Similarly, other sanitizers, such as UndefinedBehaviorSanitizer [44] detect various forms of undefined behavior, such as null pointer dereferencing, while MemorySanitizer [45] tracks the propagation of uninitialized values. Beyond general-purpose sanitizers, systems such as SoftBound [199] (for spatial safety) and CETS [200] (for temporal safety) associate metadata with each pointer in protected memory regions, offering near-complete protection under recompilation while maintaining backward compatibility. Similarly, Baggy Bounds [201] enforces spatial safety by rounding allocations to the nearest power of two and verifying pointer bounds at runtime through instrumentation. Recent efforts have focused on improving the efficiency of sanitizers, as seen in SanRazor [202], ASan [30], and GiantSan [203], and extending the instrumentation to unsafe regions of otherwise memory-safe languages, as with RustSan [204] for Rust. In contrast to purely software-based enforcement mechanisms, co-design approaches such as WatchdogLite [31] and HWASan [32] investigate hardware-accelerated, compiler-assisted pointer validation. These techniques exploit ISA extensions that cooperate with compiler-generated instrumentation to propagate pointer metadata and delegate memory-safety checks (e.g., pointer validation) to hardware-assisted operations rather than relying exclusively on software. This demonstrates how compiler instrumentation can be synergistically integrated with hardware support to achieve improved performance with lower overhead.

3.1.3.3 Runtime Enforcement Mechanisms

Low-level languages such as C and C++ expose raw pointers and manual memory management mechanisms, and do not enforce memory safety by default. Programmers work directly with raw pointers and must manually allocate and free memory, which leaves systems vulnerable to spatial and temporal errors [188]. In contrast, managed runtime systems enforce memory correctness dynamically through automatic Garbage Collection (GC), type safety, and bounds checks, preventing large classes of errors at execution time [205, 206]. Modern languages such as Java and C# automatically track object lifetimes, manage references, and eliminate explicit deallocation, thus removing entire categories of temporal memory vulnerabilities. Examples of such collectors include ZGC [207], Shenandoah [208], and Azul C4 [209], which use concurrent tracing (e.g., using load/store barriers), compaction and relocation to minimize memory fragmentation and pause time (or stop-the-world-phases) to synchronize the global memory view of the applications with the GC. For latency-critical systems, IBM Metronome and its successors demonstrated predictable real-time GC [210, 211]. Beyond conventional managed languages, researchers have investigated how to bring runtime memory safety guarantees to C and C++. The Boehm–Demers–Weiser conservative GC for C/C++ achieves automatic memory management without changing language semantics [212, 213], reducing bugs while accepting precision trade-offs. Complementary research has explored the feasibility of integrating GC into future C++ standards [214], providing real-time GC for safety-critical Java systems [215] and optimizing safe paths by eliminating bounds checking at runtime [216, 217].

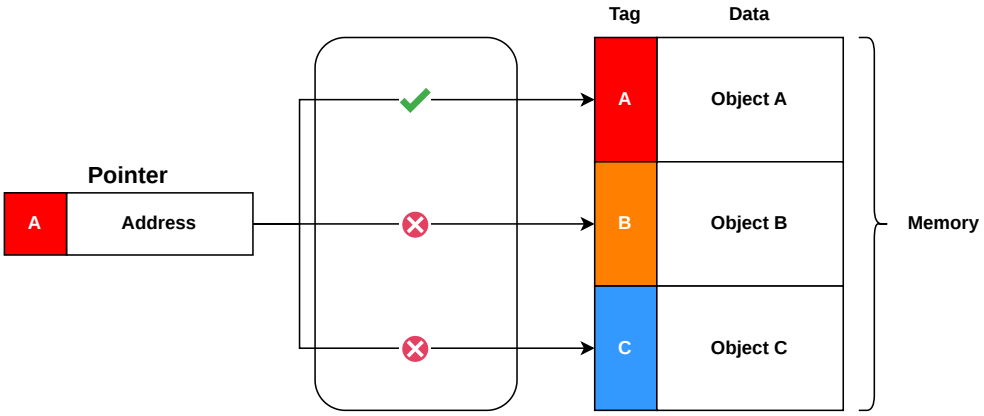


Figure 10: Overview of tagged memory mechanisms, where metadata (tags) are associated with memory regions or pointers to enforce memory safety at runtime.

3.1.3.4 Hardware-enforced Memory Safety

In recent years, mainstream ISAs have been integrating hardware mechanisms to provide memory safety with low runtime overhead. These mechanisms reduce the need for compiler-inserted instrumentation and rely on specialized hardware to provide strong spatial and temporal memory-safety guarantees at runtime. Hardware-assisted memory safety techniques are broadly divided into two categories: (i) deterministic and (ii) probabilistic, depending on how strictly they ensure protection against memory safety violations. Deterministic approaches provide precise memory-safety enforcement using hardware-checked metadata, including bounds metadata, fat pointers, or architectural capabilities, as in HardBound [218], low-fat/fat-pointer approaches [46–49], CHERI [37], and Mondrian [219]. In contrast, probabilistic techniques detect memory-safety violations statistically, relying on randomization, partial metadata checks, and memory tagging to identify invalid accesses with high confidence but without absolute trust. For example, Arm Memory Tagging Extension (MTE) [33, 220] associates metadata tag bits with memory regions and pointers, and compares these tags on every memory access to detect mismatches that may indicate spatial or temporal memory-safety violations (see Figure 10). RISC-V has also been developing related support through pointer masking and memory-tagging extensions. The pointer masking extension [221] enables the use of unused upper address bits to store pointer metadata by applying an “ignore” transformation during memory accesses, allowing tagged pointers to be used without affecting address translation. Meanwhile, the RISC-V memory tagging extension (Zimt) [34] associates tags with memory regions and pointers, and enforces tag checks on load and store operations by comparing pointer tags against memory-resident tags, enabling detection of both spatial and temporal violations. Although these systems achieve high coverage with low performance overhead, they remain susceptible to tag collisions, where two unrelated objects share the same tag, allowing some invalid accesses to go undetected. Other probabilistic methods exploit similar principles using cryptographic extensions to provide spatial and temporal memory safety. PACMem [35], CryptSan [36], and PTAuth [222] leverage Pointer Authentication Codes (PACs) to

cryptographically sign and verify pointers, detecting corruption with a small probability of collision. MultiTag [223] extends the tagging model by combining fine-grained object-level and coarse-grained page-level tags to improve spatial coverage while preserving compatibility. HashTag [224] introduces hash-based integrity protection for tagged architectures, further mitigating tag-forgery attacks. Complementary mechanisms, such as Always-On Heap Safety (AOS) [225], leverage Arm Pointer Authentication [226] to embed a PAC into unused pointer bits, thus implicitly associating metadata with the pointer itself. A dedicated microarchitectural logic unit enforces authenticated memory accesses by using the pointer PAC as an index in a hardware-resident bounds table, from which the corresponding pointer bounds are retrieved and validated, thus obviating the need for explicit metadata propagation.

3.1.3.5 Formal Methods and Verified Toolchains

Formal verification is an orthogonal approach to achieving memory safety by establishing correctness through formal reasoning, rather than relying on dynamic checks or compiler instrumentation. Formal methods employ rigorous, logic-based verification and proof assistants to guarantee that program behavior conforms to its formal specification. From a compiler perspective, the CompCert C compiler [227] is a landmark example of a formally verified toolchain. Developed and proven correct using the Coq proof assistant, CompCert guarantees that if a C program is memory-safe under its formal semantics, the compiled binary will preserve that property exactly, rather than detecting or preventing memory-safety violations already present in the source code. This ensures that verified programs remain correct after compilation and that the compiler itself introduces no new safety errors, establishing a trusted compilation chain for formally verified software. Comparable initiatives, including CreLLVM [228] and Vellvm [229], leverage formal verification techniques in intermediate representations (LLVM IR) to ensure that compiler optimizations preserve the original program behavior.

Beyond verified compilation, static verification frameworks such as Frama-C [42], VeriFast [230], and VCC [231] perform deductive program verification directly on the source code. In these approaches, developers annotate programs with formal specifications in the form of preconditions, postconditions, and invariants that characterize correct memory usage and other safety properties. Automated theorem provers and Satisfiability Modulo Theories (SMT) solvers are then employed to discharge the resulting verification conditions, establishing that all program executions conform to the user-provided specifications without requiring any runtime instrumentation. For example, Frama-C [42] offers a modular analysis platform with plugins for automatic annotation to detect undefined memory errors (e.g., invalid pointer dereference). However, these systems are critically dependent on the completeness and accuracy of the annotations and typically exhibit limited scalability when applied to larger codebases. Furthermore, formal verification has also been applied to ISAs to verify architectural correctness and consistency with their official specifications in architectures such as CHERI, RISC-V, and Armv8-A using Sail [232, 233]. At the system-level, verified operating systems such as seL4 [73] and CertiKOS [234] extend these principles to

kernel isolation and concurrency, ensuring that memory protection and functional correctness are maintained throughout the software stack. Together, these efforts share a common goal of achieving formally verified end-to-end memory safety, spanning from the ISA to the entire system stack.

3.1.3.6 Static Analysis and Dynamic Binary Instrumentation

Although memory-safe languages and compiler instrumentation offer strong guarantees, most systems software written in C/C++ continues to rely on analysis tools to uncover memory safety vulnerabilities during development and testing. In general, these techniques fall into two categories: (i) static analysis tools and (ii) dynamic analysis tools. Static analysis tools inspect source code or intermediate representations without executing the program. These tools typically use techniques such as abstract interpretation [235], separation logic [236], symbolic execution, and constraint solving to reason abstractly about memory behavior, pointer states, and control-flow across all possible paths of execution. While static analyzers may produce false positives due to their conservative approximations, they introduce no runtime overhead and are scalable to large codebases. This makes them ideal for early integration into secure software development pipelines. Several widely adopted static analyzers exemplify this approach. Clang Static Analyzer [237], part of the LLVM ecosystem, applies path-sensitive symbolic execution to explore different code paths that may lead to memory-safety violations. Cppcheck [27] performs lightweight, flow-sensitive analysis, and is particularly popular in embedded and real-time systems due to its low analysis cost. Infer, developed by Meta, performs scalable static analysis using abstract interpretation with bi-abductive separation logic and compositional reasoning, and has been deployed in large production codebases at Meta to automatically detect memory errors without requiring user-provided specifications [236]. Industrial tools such as Coverity [28] and Semmle/CodeQL [41] have demonstrated strong effectiveness in detecting bugs in complex software such as the Linux kernel [29, 238]. In contrast to static techniques, dynamic analysis observes program behavior at runtime, identifying violations on the specific execution paths exercised during testing. These tools instrument the compiled binary at runtime or observe memory interactions directly to detect violations. Representative examples include Valgrind [239] and Dr. Memory [240], which track memory allocations, pointer accesses, and object lifetimes using shadow memory.

3.1.3.7 Fuzzing and Automated Testing

Fuzzing [241] is a well-established security technique that involves generating large quantities of synthetic inputs to systematically explore unexpected execution paths and uncover memory-safety or other vulnerabilities [242]. Modern greybox fuzzers use lightweight instrumentation or hardware-assisted tracing mechanisms to enhance execution coverage and systematically steer input generation toward previously unexplored program paths, thereby achieving substantially higher effectiveness than earlier unguided fuzzing techniques. Representative fuzzing engines include AFL [242], AFLFast [243], AFL++ [244], Angora [245], Honggfuzz [246], libFuzzer [247], FairFuzz [248], and MOpt-AFL [249], as well as hybrid approaches such as SymCC-AFL [250] that integrate greybox fuzzing with symbolic execution. These systems typically

rely on address sanitizers, such as ASan [43], to detect memory safety violations during fuzzing campaigns. For example, binary-oriented sanitizing fuzzers such as QASan [251] and QMSan [252] perform dynamic binary instrumentation at runtime, approximating the behavior of compiler-based sanitizers without requiring recompilation of the original source code. QASan [251] works by dynamically instrumenting executables within Quick Emulator (QEMU) and maintaining ASan-like shadow memory to detect spatial and temporal heap violations. In contrast, QMSan [252] focuses on detecting uninitialized memory usage using a selective multi-layer instrumentation strategy that reduces overhead through opportunistic re-execution. In these designs, the sanitizer provides the violation detection logic, while the fuzzer drives input exploration. ParmeSan [253] follows a sanitizer-guided approach, using sanitizer runtime checks as feedback to steer input generation and improve bug coverage. Furthermore, systems such as MEMFuzz [254] further enhance fuzzing effectiveness by incorporating memory-access feedback to guide mutation strategies toward memory-intensive or security-relevant code paths. Recently, hardware-level fuzzing has emerged as a promising direction to detect memory safety issues in large-scale processor and SoC designs. By directly stimulating microarchitectural components, such approaches facilitate the identification of memory-related vulnerabilities that may elude conventional software-based detection techniques [255]. These techniques target hardware subsystems such as caches and MMUs, exposing memory-safety issues caused by microarchitectural and architectural behavior. In parallel, hardware-assisted fuzzing techniques leverage processor-supported mechanisms to improve fuzzing efficiency at the software level. Frameworks such as kAFL [256] utilize hardware tracing features (e.g., Intel Processor Trace) to efficiently fuzz kernel-level code without heavy instrumentation overhead. Together, these approaches illustrate the growing integration of hardware support in automated vulnerability discovery.

3.1.3.8 Summary

Table 5 summarizes the main memory-safety enforcement and mitigation techniques discussed in this section, following the taxonomy introduced in Figure 8. For each category, we highlight representative works, key features, and their overall pros and cons. As depicted in the table, these approaches exhibit distinct trade-offs. Techniques such as memory-safe languages and hardware capability systems provide strong spatial and temporal guarantees, but often require significant changes to the software stack or underlying architecture. Similarly, formal methods offer strong correctness guarantees but face scalability and adoption challenges in large systems. In contrast, compiler-based instrumentation, dynamic analysis, and fuzzing provide practical and widely deployable solutions, enabling effective detection of memory-safety violations with minimal changes to existing codebases. However, these approaches typically incur runtime overhead and do not guarantee complete coverage, as they depend on instrumentation scope or execution paths explored during testing. Hardware-assisted probabilistic techniques, such as memory tagging and pointer authentication, strike a balance by offering low-overhead protection with high detection probability, albeit without deterministic guarantees.

Category	Examples	Key Features	Pros / Cons
Memory-Safe Languages	Rust [38], Ada [39]	Strong static type systems, ownership and borrowing (Rust), compile-time and runtime safety checks.	+ Strong safety guarantees in safe subsets. – Requires codebase migration; unsafe code blocks and interactions with other languages remain potential sources of memory-safety bugs.
Compiler-Based Instrumentation	ASan [43], UBSan [44], SoftBound [199], CETS [200], HWASan [32], WatchdogLite [31]	Compiler-inserted checks, shadow memory, redzones, pointer metadata, and hardware-assisted validation.	+ High bug-detection coverage and deployable during testing. – Runtime overhead, requires recompilation, and coverage limited to instrumented code and executed paths.
Runtime Enforcement	ZGC [207], Shenandoah [208], Boehm GC [212]	Garbage collection, bounds checks, type safety, automatic lifetime management, and conservative memory reclamation.	+ Eliminates many temporal errors and reduces manual memory-management bugs. – Runtime overhead, possible nondeterminism, pause-time concerns, and incomplete guarantees for conservative GC.
Hardware-Enforced Memory Safety	CHERI [37], HardBound [218], Arm MTE [33], Zimt [34], PACMem [35], PTAuth [222]	Hardware-checked bounds, permissions, memory tags, pointer integrity, and authenticated metadata.	+ Strong deterministic guarantees for capability/bounds-based systems and low-overhead probabilistic detection for tagging/authentication. – Requires hardware/ISA support and ecosystem changes; probabilistic schemes susceptible to tag or authentication collisions.
Formal Methods	CompCert [227], Frama-C [42], VeriFast [230]	Verified compilation, formal semantics preservation, annotation-based verification, and proof of memory-safety properties.	+ Strong proof-based guarantees and no runtime overhead. – Limited language support, high verification effort, scalability challenges, and annotation burden.
Static & Dynamic Analysis	Clang SA [237], Infer [236], Valgrind [239], Dr. Memory [240]	Symbolic execution, abstract interpretation, compositional reasoning, runtime instrumentation, shadow memory, and object-lifetime tracking.	+ Useful for development pipelines; static analysis has no runtime overhead, while dynamic analysis provides precise detection on executed paths. – Static analysis may produce false positives; dynamic analysis incurs high overhead and is limited to executed paths.
Fuzzing	AFL++ [244], libFuzzer [247], QASan [251], QMSan [252]	Coverage-guided input generation combined with runtime violation detection.	+ Effective discovery of memory safety bugs and crashes. – No formal guarantees, testing-time only, and dependent on explored input coverage.

Table 5: Summary of memory safety enforcement and mitigation techniques, representative examples, key features, and trade-offs.

3.1.4 Memory Safety Hardening Techniques

Memory safety hardening techniques aim to limit exploitability even when memory bugs exist, providing defense-in-depth alongside preventive mechanisms. As shown in Figure 8, these techniques are organized into categories targeting different parts of the memory system, including stack protection, heap protection, memory layout randomization, and control- and data-flow integrity.

3.1.4.1 Stack Hardening Techniques

A classical defense against stack memory corruption is the use of *stack canaries* [257–259], where a secret value is placed adjacent to critical stack values (e.g., return address) and verified before the return of the function to detect tampering. More robust approaches isolate return addresses in a protected memory region, known as a *shadow stack* [121, 260–262]. Software-based approaches such as LLVM SafeStack and ShadowCallStack isolate control-flow data from regular local variables [260, 261], whereas hardware mechanisms like Intel Control-Flow Enforcement Technology (CET) [262] and Arm GCS [121] guarantee the integrity of return addresses at the architectural level. Complementary defenses include stack-pointer integrity mechanisms that use cryptographic authentication to protect stack and frame pointers, detect pointer corruption, and prevent stack-pivot attacks [263–265].

3.1.4.2 Heap Hardening Techniques

A substantial body of work has focused on hardening memory allocators [266–269]. Early memory allocators such as DieHard and DieHarder [266, 268] introduced randomized object placement and reuse and over-provisioning to probabilistically harden against memory threats, with the latter adding security features such as allocating objects in non-contiguous pages and overwriting memory when it is freed. Subsequent designs, including FreeGuard [270] and Guarder [271], improved performance and introduced tunable security parameters, allowing explicit trade-offs between performance, memory overhead, and security. More recent systems such as SeMalloc [272] restricts freed chunks so that they can only be reused by objects of the same type, while S2malloc [273] expands heap allocation mechanisms based on entropy with free-block canaries and layout randomization techniques, improving probabilistic resistance against UAF exploitation. Complementary frameworks such as CAMP [274] and Uriah [275] selectively apply memory protections to critical heap objects, reducing overhead compared to full instrumentation. Production-hardened allocators (e.g., Scudo) combine randomization, red-zones, and quarantine (delayed reuse) mechanisms to harden against exploitation, yet remain fundamentally probabilistic and have been bypassed in real-world deployments [276].

3.1.4.3 Address Space Layout Randomization and Memory Permission Protections

Address Space Layout Randomization. A widely used hardening method is to randomly assign initial addresses to critical process regions, such as code segment, stack, heap, and shared libraries, so

that attackers cannot easily predict where to redirect the control flow or corrupt data. By forcing adversaries to guess these addresses, Address Space Layout Randomization (ASLR) substantially increases the difficulty of conducting Return-Oriented Programming (ROP) and data-oriented attacks [277]. However, early evaluations on 32-bit systems demonstrated that entropy was limited and that memory layouts often remained partially predictable [278]. To address these limitations, subsequent designs proposed strengthening ASLR through increased entropy and finer-grained randomization [279–281]. Despite these improvements, ASLR remains a fundamentally probabilistic defense and has been repeatedly shown to be vulnerable to bypass techniques [282]. Empirical studies have revealed that real-world implementations of ASLR often provide significantly less entropy than assumed, exhibiting correlations and predictability that create an illusion of randomness and facilitate reliable exploitation [283]. Kernel-level randomization has also been shown to be vulnerable to side channels and information leaks, which can undermine the effectiveness of Kernel Address Space Layout Randomization (KASLR) in practice [284, 285].

Memory Permission Protections. Memory permissions enable the specification of security policies that limit memory operations (read, write, execute) within specific memory regions. Classical techniques, such as Write XOR Execute (W⊕X), prevent pages from being writable and executable at the same time. This thwarts straightforward code-injection attacks and pushes adversaries toward code-reuse methods such as ROP [286]. A stricter variant of this technique is Execute-Only Memory (XOM), which strips read permissions from code pages to block gadget harvesting and frustrate Just-in-Time (JIT)-ROP and data-exfiltration attacks. Implementations like NORAX [287] for AArch64 and uXOM [288] for ARM Cortex-M microcontrollers demonstrate that such mechanisms can be enforced with low overhead while remaining backward compatible. Although execute-only memory prevents code reads, empirical studies show that microarchitectural side channels can still leak information about execute-only code, limiting the effectiveness of XOM as a standalone defense [289].

3.1.4.4 Control-Flow, Pointer, and Data-Flow Integrity

A substantial subset of memory-safety vulnerabilities does not rely on injecting new code, but instead seeks to corrupt existing control-flow or data-flow dependencies in order to deviate program execution from its intended semantics. To mitigate such attacks, an extensive body of integrity-enforcement mechanisms has been proposed that constrain the runtime evolution of pointers and data values.

Control-Flow Integrity. CFI is a defensive technique that restricts control-flow transfers (e.g., indirect calls, jumps, and returns) to a set of valid statically or dynamically computed targets, mitigating code-reuse attacks such as ROP and Jump-Oriented Programming (JOP) [290–292]. Binary-level approaches enforce these constraints directly on executables using static or dynamic analysis, making them suitable for legacy software, but often rely on coarse-grained approximations due to limited semantic information [293]. Compiler-based mechanisms, such as LLVM-CFI, leverage type and control-flow information to enforce finer-grained constraints with stronger guarantees, at the cost of requiring recompilation [294]. Dynamic refinements further improve precision by adapting target sets based on observed execution

paths [295]. Hardware-assisted mechanisms, including Intel CET and Arm Branch Target Identification (BTI), provide efficient enforcement of forward- and backward-edge protections with low overhead [121, 262]. Complementary systems such as TypeArmor [296] and VTrust [297] refine target precision and protect virtual calls, while hybrid designs combining CFI with shadow stacks further strengthen return-address integrity [298]. Despite limitations in precision and susceptibility to information leaks, CFI remains a widely deployed and effective control-flow hardening mechanism.

Pointer and Code-Pointer Integrity (PIE / CPI). Pointer Integrity (PIE) and Code-Pointer Integrity (CPI) mechanisms aim to prevent arbitrary pointer (data or code) reuse, forgery, corruption, or reuse in operations not intended by the original program flow. While traditional CFI restricts where execution may transfer by examining control-flow graphs, PIE and CPI validate pointers that can participate in those transfers and therefore prevent control-flow hijacking. Kuznetsov et al. [299] introduced CPI as a mechanism to enforce memory safety for code pointers. This approach employed static analysis to identify all memory objects containing code pointers and relocate them to protected memory regions. These safeguarded regions are isolated from the standard process address space and can be accessed exclusively via operations that are either statically proven safe at compile time or are subject to dynamic safety checks at runtime. Complementary hardware extensions such as PAC [300] support CPI at the ISA level by cryptographically signing pointer values with a per-process key. Upon dereference, the pointer authentication code is verified by hardware, detecting tampering or unauthorized reuse. Building on this model, Kim et al. [301] improved conventional CFI by combining cryptographic pointer authentication with static and dynamically refined control-flow enforcement. In this scheme, each function pointer is associated with a Message Authentication Code (MAC) that is cryptographically bound to its corresponding CFI label, thus guaranteeing the integrity of the pointer while simultaneously enforcing compliance with a dynamically constrained control-flow graph.

Data-Flow Integrity (DFI). Data-Flow Integrity (DFI) enforces data-flow integrity by restricting which instructions are authorized to write to specific memory objects. By instrumenting memory writes and validating them at runtime against a precomputed data-flow graph derived at compile time using static analysis, DFI prevents both control-flow hijacking attacks and data-only attacks that corrupt program state without modifying control-flow [302]. Early work on DFI demonstrated the feasibility of enforcing non-control-data protection, but incurred significant runtime and memory overhead due to pervasive instrumentation of memory writes and maintenance of per-object metadata [302]. WIT [303] implemented a compile-time "coloring" mechanism in which authorized store instructions and memory objects are annotated with corresponding, mutually consistent color identifiers. During program execution, each write operation is validated by checking that the color associated with the issuing instruction matches the color of the target memory object, thereby enforcing the integrity constraints of memory writes. Hu et al. [304] demonstrated that non-control-data corruption can bypass CFI and other control-flow defenses, reinforcing the need for data-flow protection. In parallel, Zhao et al. [305] systematically analyzed the sources of DFI overhead and proposed optimization strategies to improve its practicality in real systems.

3.1.4.5 Summary

Category	Examples	Key Features	Pros / Cons
Stack Hardening	StackGuard [257], ProPolice [258], SafeStack [260], ShadowCallStack [261], Intel CET [262], Arm GCS [121]	Stack canaries, separation of unsafe stack objects, shadow stacks, and architectural protection of return addresses.	+ Effective against many stack-smashing and return-address corruption attacks. – Limited protection against non-control-data attacks, information leaks, and attacks that bypass or corrupt unprotected stack objects.
Heap Hardening	DieHard [266], DieHarder [268], FreeGuard [270], Guarder [271], SeMalloc [272], S2malloc [273], CAMP [274], Uriah [275], Scudo	Randomized allocation and reuse, memory over-provisioning, non-contiguous placement, delayed reuse/quarantine, type-based allocation, and selective protection of critical objects.	+ Mitigates heap overflows and UAF exploitation while remaining largely compatible with existing software. – Mostly probabilistic; incurs memory/performance overhead and may still be bypassed by advanced exploitation techniques.
Memory Layout Randomization	ASLR [277], fine-grained ASLR [279–281], KASLR bypasses [282, 284, 285]	Randomizes the placement of code, stack, heap, libraries, and kernel regions to make target addresses harder to predict.	+ Low overhead and widely deployed as a first-line hardening mechanism. – Probabilistic; effectiveness depends on entropy and can be undermined by information leaks, side channels, and layout correlations.
Memory Permission Protections	W⊕X [286], XOM [287–289]	Enforces page-level access permissions, such as non-writable executable memory and execute-only code pages.	+ Prevents straightforward code injection and reduces code disclosure/gadget harvesting. – Does not prevent code-reuse attacks by itself and may be weakened by side channels or disclosure vulnerabilities.
Control-Flow Integrity	CFI [291, 292], LLVM-CFI [294], TypeArmor [296], VTrust [297], Intel CET [262], Arm BTI [121]	Restricts indirect branches, calls, jumps, and returns to valid targets using static analysis, compiler instrumentation, or hardware support.	+ Strong mitigation against control-flow hijacking, including ROP and JOP, especially with precise policies or hardware support. – Precision depends on the analysis; coarse-grained policies and information leaks may still allow valid-target attacks.
Pointer and Code-Pointer Integrity	CPI [299], PAC [300], Code-Pointer Tagging [301]	Protects sensitive pointers using isolation, protected metadata, cryptographic authentication, or message authentication codes.	+ Mitigates pointer corruption, substitution, forgery, and unauthorized reuse; complements CFI by protecting the pointers used in indirect transfers. – Often protects only selected pointer classes; requires compiler/runtime or hardware support and may face compatibility or key-management challenges.
Data-Flow Integrity	DFI [302], WIT [303], data-oriented attack studies [304], DFI optimizations [305]	Restricts which instructions may write to specific memory objects using static data-flow analysis, runtime checks, and metadata.	+ Protects non-control data and complements CFI/CPI against data-oriented attacks. – Typically incurs high runtime and memory overhead, requires instrumentation, and faces scalability/precision challenges.

Table 6: Summary of memory-safety hardening techniques, representative examples, key features, and trade-offs.

Table 6 summarizes the main memory-safety hardening techniques discussed in this section, following the taxonomy presented in Figure 8. The table highlights representative works, key mechanisms, and their pros and cons. In general, these techniques provide probabilistic protection and are designed to increase the difficulty and cost of exploitation rather than guaranteeing full memory safety. For instance, heap hardening techniques introduce randomness, isolation, and delayed reuse strategies to mitigate common exploitation primitives such as heap overflows, at the cost of additional memory overhead. Similarly, address space layout randomization and memory permission protections (e.g., XOM) provide low-overhead defenses that are widely deployable, but rely on entropy and can be bypassed using side-channel attacks. CFI and CPI offer stronger protection against control-flow hijacking by constraining indirect branches and

protecting pointer validity. However, their effectiveness depends on the precision of the analysis and may require compiler support or special hardware extensions. DFI complements these approaches by protecting non-control data, but typically incurs higher overhead due to fine-grained instrumentation and metadata tracking. Overall, hardening techniques provide an essential layer of defense-in-depth and are widely deployed in production systems. However, their probabilistic nature reinforces the need for stronger enforcement mechanisms, motivating the integration of hardware-assisted solutions that can provide deterministic guarantees with acceptable overhead.

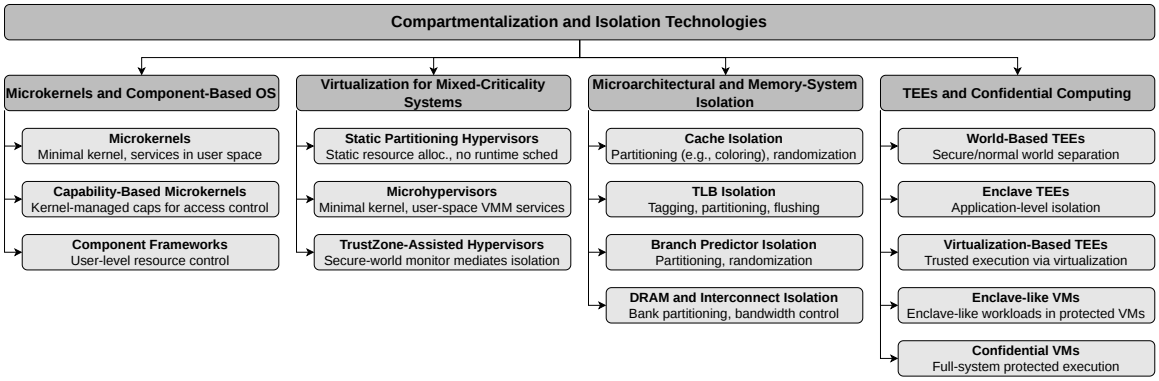


Figure 11: Hierarchical taxonomy of compartmentalization and isolation technologies for MCS.

3.2 Compartmentalization and Isolation Technologies

This section reviews the state of the art in compartmentalization and isolation technologies in the context of embedded systems and MCS. The primary objective of these technologies is to limit the impact of security breaches by enforcing strict separation between execution domains and by imposing well-defined interaction interfaces, rather than assuming that all software components within a system are mutually trustworthy. Over the past decades, a broad spectrum of isolation mechanisms has emerged, encompassing OSs, microkernels, RTOSs, hypervisors, TEEs, and system-wide isolation technologies. Figure 11 summarizes the taxonomy followed in this chapter, organizing these approaches into four main categories: (i) microkernels and component-based operating systems, (ii) virtualization for MCS, (iii) microarchitectural and memory-system isolation, and (iv) TEEs and confidential computing.

3.2.1 Microkernels and Component-based Operating Systems

Microkernels [306] differ fundamentally from traditional monolithic kernels [307] by implementing only a minimal set of essential mechanisms, such as scheduling, IPC, and basic memory management, within the kernel itself and running higher-level services, including device drivers, file systems, and network

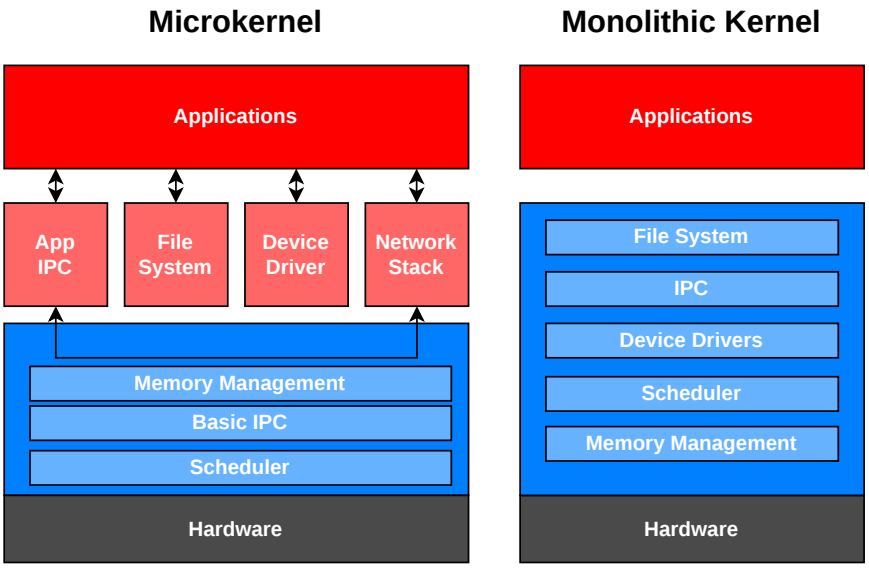


Figure 12: Comparison between monolithic and microkernel architectures. In monolithic kernels, all core services (e.g., drivers, file systems, networking) execute in kernel space, whereas microkernels minimize the trusted kernel to essential mechanisms and move services to user space, communicating via controlled interfaces such as IPC.

stacks, as user-space applications. As illustrated in Figure 12, isolation between components in microkernels is typically enforced using distinct address spaces, and interactions are mediated through kernel-defined mechanisms. This architectural property enables transparent interposition of security monitors and the enforcement of fine-grained system-wide security policies without requiring modifications to the application logic. As a result, microkernels offer strong fault containment, controlled information flow, and improved system robustness. Furthermore, this decomposition of the OS into a set of mutually isolated components significantly reduces the TCB, improves modularity and reduces certification and verification costs, properties that are particularly valuable in safety and security-critical systems [308].

Capability-based microkernels are the most widely deployed microkernels. In these systems, capabilities govern access to system resources by providing unforgeable keys to system objects that are managed by the kernel. Early systems such as EROS [68] established the foundational concepts of capability-based operating systems and strongly influenced modern microkernel designs, including seL4 [51], Fiasco.OC [53], and Composite [309]. These systems mainly target contexts that require high assurance, strong isolation, and reduced TCB, including mixed-criticality and safety-certified environments. seL4 is a capability-based microkernel designed to provide strong spatial and temporal isolation with a rigorously minimized TCB [73]. All access to memory, IPC endpoints, and kernel objects (e.g., virtual address-space objects) is mediated by kernel-managed capabilities. seL4 is distinguished by its machine-checked formal proofs of functional correctness, authority confinement, and information-flow security, guaranteeing that the kernel enforces its specification exactly. Beyond functional correctness, seL4 supports

timing and scheduling analysis, enabling reasoning about temporal isolation in mixed-criticality systems. Deployment frameworks such as CAmkES [310] and platform abstractions such as the seL4 Core Platform [311] facilitate the construction of statically composed, analyzable systems suitable for certification. Fiasco.OC [52] continues the philosophy of L4 microkernel design as a capability-based microkernel paired with the L4 Runtime Environment, which provides primitives for resource management and delegates their management policies for user-level applications, allowing flexible system construction. A study conducted by Lampka et al. [53] has shown the practical viability of using microkernel-based systems built on Fiasco.OC/L4Re that align with automotive safety-certification standards. Composite OS [309] explores fine-grained component isolation combined with hierarchical user-level scheduling and resource management, targeting predictable performance and scalable execution on multicore systems. By moving scheduling and resource control out of the kernel, Composite enables flexible system policies while maintaining strong isolation between components.

3.2.2 Virtualization for Mixed-Criticality Systems

Virtualization is widely used to consolidate workloads of different security, safety, and criticality levels while preserving strong spatial isolation, predictable execution, and certification readiness. Unlike cloud-oriented hypervisors, these systems emphasize static resource allocation, minimal TCB, and freedom-from-interference execution, according to safety standards such as ISO 26262 [21], DO-178C [196], IEC 61508 [197], and EN 50128 [198]. Many modern hypervisors, particularly those targeting embedded and safety-critical domains, adopt the separation kernel philosophy [312], which enforces strong spatial and temporal isolation through minimal trusted components and controlled communication channels. SPHs realize this model through strict resource allocation at boot time and by avoiding resource sharing at runtime. Alternative approaches also aim to provide similar isolation guarantees, including microhypervisors, which minimize kernel functionality and delegate services to user space, and TrustZone-assisted virtualization approaches, which rely on hardware-enforced secure execution domains. The following discussion highlights representative systems across both industrial and research ecosystems.

Commercial Safety-oriented RTOS and Hypervisors. Commercial RTOS-based hypervisors such as Helix [313], PikeOS [124], INTEGRITY Multivisor [314], and QNX Hypervisor [126] target critical safety domains of avionics, automotive, and industrial control. These RTOSs typically combine a partitioning hypervisor with optional RTOS services by leveraging hardware virtualization extensions to enable isolation between VMs, allowing mixed-critical consolidation under a strictly partitioned execution model. These often use paravirtualization to share devices and to provide system services. Although highly suitable for certification, isolation is enforced at the partition level, and software within a partition is assumed to be mutually trusted. Others, such as LynxSecure [315], follow a stricter pure separation kernel design, providing a minimal isolation layer and avoiding conflating RTOS services with the partitioning layer, promoting a more flexible, modular, and distributed architecture. These commercial systems enforce isolation at the partition level and are well-suited for certification but generally assume mutual trust among

all software components within a partition. Standalone safety-certified RTOSes, such as VxWorks [316], INTEGRITY [317], QNX Neutrino [318], and LynxOS [319], are designed primarily to guarantee strong temporal predictability. They achieve this by employing preemptive priority-based scheduling in combination with time partitioning mechanisms to bound interference across software partitions.

Open-Source and Research Hypervisors. Multiple open-source and research systems have similar design principles with varying degrees of minimality and flexibility. For instance, the NOVA microhypervisor [320] minimizes its TCB by decomposing the hypervisor into a small kernel-level component and user-level VMM and hypervisor services, delegating device emulation and hardware virtualization to unprivileged services. seL4 [51] follows a similar microkernel-based approach, where the kernel provides only isolation and minimal virtualization primitives, and virtualization functionality is implemented in user-level VMMs, typically instantiated per VM. Open-source embedded hypervisors such as Jailhouse [132] and Bao [118] both implement SPH designs that eliminate resource sharing and runtime scheduling. However, Jailhouse requires a Linux host for system initialization and management, making Linux part of the TCB. In contrast, Bao provides a minimal virtualization layer designed specifically for mixed-criticality embedded systems, emphasizing deterministic execution, minimal interference, and a codebase amenable to certification. Bao also provides an alternative architectural design specifically tailored to satisfy the stringent timing guarantees required in real-time and safety-critical systems, while avoiding non-deterministic latencies introduced by MMU-based address translation (e.g., page table walks) [141]. This design is based on the emerging paradigm of *MMU-less virtualization*, which leverages MPU-based protection mechanisms (see Background Chapter 2). Santos et al. [138] added support for this model to Bao targeting Armv8-R, Infineon Aurix TC4x, and Renesas RH850 (RISC-V MMU-less with M+HS + VS+U). Other space grade-Type-1 hypervisors, such as Xtratum [15], employ static partitioning, device direct control, and para-virtualization techniques to implement hypervisor services, thus increasing predictability. Other safety hypervisors, such as Quest-V [321], occupy an intermediate position in this design space. It adopts a multikernel-inspired architecture in which multiple kernel instances execute on separate cores and are isolated using hardware virtualization support (e.g., Intel VT-x). A lightweight monitor layer mediates inter-kernel communication and recovery. This design improves fault containment by confining failures to individual kernels and enables static resource allocation for improved predictability. Other general-purpose hypervisors, such as Xen [125], have been adapted for mixed-criticality systems through static partitioning and safety-oriented configurations. In its traditional deployment, Xen relies on a privileged control domain (Dom0), which typically runs Linux, which is part of the TCB. However, recent Dom0-less configurations [125] remove this dependency by booting all domains directly from the hypervisor.

TrustZone-Assisted Hypervisors. LTZVisor [322], μRTZVisor [323], and the work by Pinto et al. [324] exploit Arm TrustZone to implement lightweight virtualization by placing a minimal monitor in the secure world that mediates access to hardware resources and interrupts for non-secure software. LTZVisor establishes the basic TrustZone-based virtualization model, in which a secure world execution context hosts a small trusted monitor and selected real-time components that make up the TCB, while general-purpose

software runs in the non-secure world. μRTZVisor refines this model with a microkernel-like design with software-defined capability abstractions to mediate access to system resources. In contrast, Pinto et al. extend the TrustZone-assisted approach to multicore platforms, allowing asymmetric execution of guest OSs across cores. While these systems achieve low overhead, the TrustZone two-world abstraction and centralized secure-world control fundamentally limit scalability and fine-grained isolation. This class of approaches gained particular relevance prior to the widespread availability of Arm VE, serving as a practical alternative for enabling isolation on platforms lacking native virtualization support. However, with the introduction of hardware-assisted virtualization in modern Arm processors, TrustZone-assisted hypervisors have seen limited adoption for general-purpose virtualization, remaining primarily relevant in constrained or legacy embedded environments.

3.2.3 Microarchitectural and Memory-System Isolation

Architectural isolation mechanisms such as MMU and privilege separation provide strong software-level isolation. However, they do not prevent interference stemming from shared hardware microarchitectural resources, including caches, TLBs, branch predictors, speculative execution units, DRAM controllers, and on-chip interconnects. These shared structures constitute a significant source of execution-time variability and microarchitectural side channels [325], as evidenced by transient-execution attacks such as Spectre [326]. Unregulated sharing of these components can compromise both security properties and real-time timing guarantees. Consequently, an extensive body of research has concentrated on microarchitectural and memory system isolation techniques designed to complement and strengthen conventional protection mechanisms. In this section, we survey approaches for (i) cache isolation, (ii) TLB isolation, (iii) branch predictor isolation, and (iv) DRAM and interconnect isolation. Table 7 summarizes representative techniques across the different microarchitectural layers, highlighting their target resources, underlying approaches, and primary objectives.

Cache Isolation. Caches are a primary source of microarchitectural interference and timing side channels. Cache partitioning is widely used to improve timing predictability in real-time systems [351] and mitigate cache-based attacks [352] by bounding interference due to evictions. A common strategy dedicates distinct portions of the LLC to different protection domains [353], either through software techniques such as page coloring, which control address mapping to avoid cache-set overlap [327], or hardware-assisted mechanisms that provide finer-grained and lower-overhead isolation. Hybrid approaches such as CATalyst [328] combine both, partitioning the LLC into secure and non-secure regions, while systems like Cachelets [329] dynamically allocate cache capacity to enclaves. However, strict partitioning can reduce cache utilization and hinder data sharing in MCS. To address this, previous work has proposed analyzable interference channels [354] and cache organizations that enable predictable sharing with bounded interference [355]. Other approaches have directly modified cache designs, such as DAWG [330], to limit information leakage in speculative processors. Complementary techniques have relied on randomization (e.g., CEASER [331]) to disrupt attacker-controlled cache mappings. Finally, software-hardware co-design

Category	Technique	Description	Goal
Cache Isolation	Page Coloring [327]	Software partitioning of LLC sets through address-to-cache-set mapping control	Predictability + Security
	CATalyst [328]	Hybrid hardware/software LLC partitioning	Predictability
	Cachelets [329]	Dynamic LLC capacity allocation for isolated domains	Security
	DAWG [330]	Hardware-enforced cache isolation domains across the cache hierarchy	Security
	CEASER [331]	Randomized cache indexing to disrupt attacker-controlled cache mappings	Security
	Cache-aware / real-time allocators [332–335]	Cache-aware placement, LLC-bank bandwidth regulation, and bounded-time allocation	Predictability
TLB Isolation	ASID/VMID tagging	Tagged TLB entries to reduce flushes and separate address spaces/domains	Predictability + Security
	Page coloring for TLB [336]	Software partitioning of TLB sets through controlled page allocation	Predictability
	Secure TLB [337]	Domain-based TLB tagging to prevent cross-domain eviction and probing	Security
	Dynamic allocation [338]	Hardware/software co-design for predictable TLB-entry allocation	Predictability
	Randomized replacement [339]	Randomized TLB fill and replacement policies	Security
Branch Predictor	IBRS/STIBP [340]	Predictor restriction/isolation across contexts or hardware threads	Security
	HyBP [341]	Hybrid branch-predictor isolation combining physical separation and randomization	Security
	Encrypted predictors [342]	Predictor-state obfuscation through encryption/randomization	Security
	STBPU [343]	Dedicated branch-predictor structures to avoid cross-domain sharing	Security
	Predictor flushing [344]	Temporal isolation by flushing predictor state during context switches	Predictability
DRAM & Interconnect	MemGuard [345]	Memory-bus bandwidth reservation for cores/tasks	Predictability
	MemGuard-RW [346]	Read/write-aware memory-bus bandwidth regulation	Predictability
	BRU [347]	Hardware-based memory-bandwidth regulation	Predictability
	MemPol [348]	Memory-traffic policing for shared memory-system resources	Predictability
	AXI-REALM [349]	Traffic regulation for shared Advanced eXtensible Interface (AXI) interconnects	Predictability
	PMC [350]	Criticality-aware DRAM request scheduling	Predictability
	H-MBR [134]	VM-level memory-bandwidth reservation in virtualized systems	Predictability

Table 7: Summary of representative microarchitectural and memory-system isolation techniques.

approaches further improve predictability by shaping memory placement and access patterns. Examples include cache-aware allocators such as CAMA [332], which improve cache predictability through controlled memory placement, and per-bank bandwidth regulation mechanisms for shared LLCs [333], which mitigate cache-bank contention. Real-time allocators such as TLSF [334, 335] complement these approaches by providing bounded-time allocation and deallocation, thereby supporting worst-case execution-time analysis.

TLB Isolation. The TLB is a shared microarchitectural resource that can induce timing variability and enable covert and side channels across protection domains [356, 357]. Prior work has shown that contention and replacement behavior in the TLB can leak sensitive information or compromise temporal predictability in MCS [357]. To mitigate these effects, several approaches have proposed TLB partitioning mechanisms that provide spatial isolation by allocating disjoint translation resources to tasks or criticality domains. Contemporary ISAs already offer basic support through tagged entries (e.g., Address Space Identifiers (ASIDs), process identifiers, or VM identifiers), which reduce flushes across context switches and partially confine interference. Software-based strategies, such as [336], leveraged page coloring to control set-level contention. Architectural proposals like Secure TLBs [337] introduced domain-based tagging to prevent cross-domain eviction and probing. Other work [338] explored software–hardware co-design to dynamically assign TLB entries, enabling predictable translation behavior with flexible allocation. Complementary techniques [339] employed randomized fill and replacement policies to reduce correlation between observable access patterns and internal TLB state.

Branch Predictor Isolation. Branch predictors (e.g., BTB) are also a significant source of cross-domain interference and side-channel leakage. Predictor partitioning, flushing, or redesign has been proposed to mitigate control-flow side channels and speculative interference. As surveyed by Kim et al. [358], proposed

isolation mechanisms include partitioning of predictor state per context or privilege level [359] (e.g., Intel STIBP and IBRS [340]), randomization and encryption of the branch predictors [342, 360], hybrid approaches that merged randomization and physical isolation (e.g., HyBP [341]), or completely avoided the sharing of these microarchitectural structures (e.g., STBPU [343]). Software-only defenses such as Half&Half [361] have also been explored to isolate or obfuscate predictor state without hardware modifications. Others provided temporal isolation by flushing the predictor history on context switches [344]. For example, Wistoff et al. [344] added speculation fences that invalidate the predictor and cache state when context switching. This solution was tested on the CVA6 core using the seL4 microkernel.

DRAM and Interconnect Isolation. At the memory-system level, shared DRAM and on-chip interconnect resources remain a significant source of unpredictability and contention in MCS, exposing the attack surface to timing attacks. In particular, DRAMA [362] showed that attackers can reverse-engineer DRAM addressing to intentionally trigger bank and row conflicts across cores, creating cross-CPU timing channels even under strong software isolation. BUSted [363] demonstrated that timing variations introduced by arbitration logic at the bus interconnect allowed attackers to infer memory access patterns across isolation domains, effectively bypassing hardware-enforced protection mechanisms. Costa et al. [364] showed that I/O protection mechanisms can also introduce contention channels in MCS. These effects arise from interference in IOMMU request handling and translation caching, further broadening the attack surface to include the I/O subsystem. This motivates hardware and OS mechanisms such as bank partitioning, bandwidth regulation, and requirement-aware memory controllers. Hardware regulators such as the BRU [347], MemGuard/MemGuard-RW [345, 346], MemPol [348], and AXI-REALM [349] provided cycle- or bandwidth-level control over memory traffic to bound inter-core interference and improve timing predictability of real-time tasks. Complementary mechanisms such as IRQ coloring reduced interrupt-generated interference on shared memory paths [135]. Recent security analyses [365] showed that bank and row conflicts can be exploited to drastically increase execution times of critical workloads, motivating stronger isolation at the DRAM level. Mixed-criticality-aware memory controllers, including PMC [350], explicitly scheduled off-chip memory requests according to task criticality and enabled spatial and temporal isolation of DRAM banks. In the virtualization layer, H-MBR [134] introduced a memory-bandwidth reservation mechanism that enables hypervisors to enforce temporal isolation between multiple VMs. Finally, other studies developed DRAM-aware memory allocators for better performance and isolation [366].

3.2.4 Trusted Execution Environments and Confidential Computing

TEEs provide hardware-enforced isolated execution environments intended to protect security-sensitive code and data from a potentially compromised host OS or hypervisor. In the embedded systems domain, TEEs are often positioned as a building block to separate critical services (e.g., key management, secure boot, authentication, safety monitors) from large, complex, and frequently updated non-critical software stacks. Figure 13 provides a comparative overview of representative TEE and confidential computing architectures, highlighting their system-level organization and trust boundaries. This subsection reviews

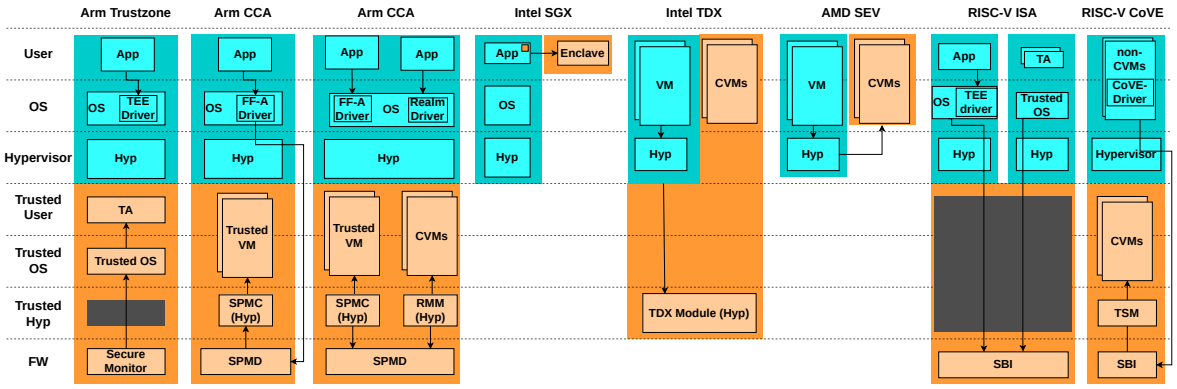


Figure 13: Comparison of representative TEEs and CCA technologies across different platforms. The figure contrasts world-based isolation (e.g., Arm TrustZone), enclave-based designs (e.g., Intel SGX), and virtualization-based approaches (e.g., Intel TDX, AMD Secure Encrypted Virtualization (SEV), Arm CCA, and RISC-V CoVE). Privilege levels are shown vertically, while trusted components and isolation boundaries highlight the evolution toward full-system protected execution using Confidential Virtual Machines (CVMs). Adapted from [367].

the dominant TEE design strategies most relevant to this thesis, spanning world-based TEEs (e.g., TrustZone [58]), enclave TEEs (e.g., Intel SGX [368]), and virtualization-based TEEs. We then connect these trends to the recent shift toward confidential computing, where full-system protected execution (i.e., confidential VMs) is used to defend workloads against untrusted infrastructure, motivating stronger hardware isolation across the CPU, memory system, and I/O subsystem.

TrustZone-assisted TEEs. Arm TrustZone technology, implemented in the Cortex-A and Cortex-M processor families [58], constituted one of the first commercially deployed hardware mechanisms for realizing TEEs in embedded systems. In this model, the system is partitioned at the hardware level into two execution domains: (i) an untrusted *normal world* that runs a general-purpose operating system and (ii) a trusted *secure world* that hosts a small trusted runtime responsible for security-critical services such as key management, cryptographic operations, authentication, and firmware verification. Transitions between these worlds are mediated by hardware and a trusted secure monitor, ensuring that code and data in the secure world remain protected even if the normal-world operating system is compromised. OP-TEE [369] is a prominent open-source reference implementation of this model targeting Application Processor Units (APUs), providing a secure runtime for trusted applications together with services such as secure storage, cryptographic primitives, and standardized APIs aligned with the GlobalPlatform specification [370]. Recent Arm platforms further extend this model through Arm FF-A [371], which standardizes communication between normal- and secure-world components and enables the deployment of multiple secure partitions, isolated using secure-world virtualization support (e.g., S-EL2). TrustZone has also motivated extensive academic research on privilege separation, real-time guarantees, reduced TCBs, and composability of secure services in MCS. For instance, RT-TEE [372] integrates TrustZone with a real-time operating system to provide predictable execution of secure applications, introducing real-time scheduling, bounded secure-world transitions, and analyzable interrupt handling for cyber-physical systems. Similarly,

MR-TEE [373] follows the same approach while requiring fewer modifications to existing normal-world applications and supporting real-time interrupts in the secure world. FreeTEE [374] modifies FreeRTOS to implement a lightweight real-time TEE that runs alongside a general-purpose OS in the normal world. Despite the widespread adoption of the TrustZone Technology, studies have highlighted several structural limitations of the dual-world model. In particular, Cerdeira et al. [61] identified fundamental challenges including a large trusted computing base, complex secure monitor logic, and difficulties in reasoning about interactions between secure and normal worlds. ReZone [8] further shows that many TrustZone deployments grant excessive privilege to secure-world software. Motivated by these scalability and composability issues, recent proposals explore multi-domain secure-world models. For example, uTango [375] introduces a lightweight multi-TEE architecture for IoT devices by moving trusted applications into isolated compartments managed by a trusted kernel, reducing the TCB and improving modularity compared to the monolithic secure-world designs.

Enclave TEEs. Other TEE models provide enclaves to run applications isolated from the OS. Intel SGX [368] and AMD SEV [376] are canonical industrial examples that support encrypted memory regions, secure execution, and remote attestation for applications running on untrusted hosts. In the research domain, Keystone [60] pioneered an open-source framework to build enclaves on RISC-V, relying on standard hardware mechanisms such as PMP extension and a minimal security monitor. Sanctum [377] co-designs hardware and software by redefining page-based isolation and introducing cache partitioning techniques to harden enclaves against side-channel attacks. TIMBER-V [378] and SPEAR-V [379] explore flexible enclave designs leveraging standard RISC-V protection mechanisms (e.g., PMP and MMU) together with tagged memory regions to support dynamic enclave creation and controlled memory sharing. For deeply embedded platforms, Sancus [380] and Sancus 2.0 [381] demonstrate ultra-lightweight enclave TEEs with minimal hardware and software TCB, tailored to microcontrollers and constrained IoT nodes. CURE [382] further generalizes the enclave abstraction by supporting multiple enclave granularities, from fine-grained subspace enclaves to multicore enclaves, using minimal hardware extensions and software-managed isolation. Aion [383] is a hybrid design that provides enclave confidentiality and integrity while also aiming to achieve strong availability guarantees through a formally verified trusted kernel running on standard hardware, addressing the lack of availability guarantees in most existing enclave systems. Other systems focus on reducing the internal attack surface of TEEs, for example, DORAMI [384] accomplishes this by decomposing trusted functionality into isolated components managed by a minimal security monitor.

Virtualization-based TEEs. Virtualization-based and hybrid TEEs implement trusted execution semantics primarily using standard hardware virtualization mechanisms rather than dedicated enclave hardware modes. AnyTEE [367] and HyperEnclave [385] exemplify this approach by constructing TEEs on top of standard virtualization support (e.g., Arm VE, Intel VT-x, and the RISC-V hypervisor extension), allowing enclave-like workloads to run inside protected virtual machines without relying on specialized enclave hardware. TEEv [386] proposes a virtualization solution for mobile platforms that allows multiple TEEs from different vendors to run on the same hardware. Similarly, Bao-Enclave [139] provides an enclave-style

TEE using hardware virtualization, allowing isolated execution of security-critical workloads in MCS.

Confidential Computing and Confidential VMs. Recently, Confidential Computing (CC) has emerged in response to the growing deployment of workloads on untrusted cloud and edge infrastructures. In contrast to classical TEEs, which primarily protect security-critical components within an otherwise trusted system, CC assumes an untrusted platform and seeks to protect entire workloads from potentially malicious operating systems, hypervisors, and infrastructure. This shift reflects a modern threat model in which users no longer control the physical machines that execute their code. To support this paradigm, recent architectures extend the isolation model from application enclaves to full-system-protected execution domains via VMs. On Arm, the CCA extension introduces *Realms*, which are hardware-isolated virtual machines capable of hosting complete guest operating systems while remaining protected from a potentially malicious hypervisor and DMA-capable devices [63]. Similarly, RISC-V CoVE defines architectural extensions for protected VM execution, enabling confidential computing for full-system workloads [64]. WorldGuard [65] proposes system-wide hardware isolation that enforces domain separation across cores, memory, and devices without relying on a single trusted hypervisor. Collectively, these architectures embody a progression from component-level TEEs toward comprehensive, system-wide security substrates that are designed to provide strong confidentiality and integrity guarantees, even under large-scale, multi-tenant, and adversarial execution conditions.

3.3 Discussion and Open Gaps

In this chapter, we examine the state of the art in techniques that provide primitives analogous to those of CHERI technology, albeit through different underlying mechanisms. These span memory-safety mitigation, enforcement, and hardening strategies, as well as compartmentalization and isolation technologies tailored for critical real-time systems. Although these approaches substantially enhance the resilience of contemporary computing platforms, they operate at distinct abstraction layers and consequently leave notable gaps unaddressed, particularly in mixed-criticality and high-assurance environments.

3.3.1 CHERI in the Memory-Safety Landscape

Table 8 summarizes the design space of memory safety techniques in six key dimensions, including spatial and temporal protection, determinism, runtime overhead, compatibility with legacy code, and granularity. The following discussion interprets these dimensions and highlights the limitations that motivate architectural approaches such as CHERI. Memory-safe programming languages such as Rust [38] and Ada [39] eliminate broad classes of memory-related defects at the language level. However, retrofitting extensive legacy C/C++ codebases remains costly and, in many cases, infeasible. Furthermore, low-level software components, including kernels, firmware, and device drivers, frequently require the use of unsafe constructs, thereby reintroducing memory safety risks at the most privileged layers of the system. Address sanitizers [43, 44] aim to detect or prevent memory violations through compiler-inserted runtime

Technique	SMS	TMS	DET	RO	C/C++ Compat.	Granularity	Key Limitations / Gaps
Safe languages	✓	✓	✓	Low	△	Object-level	Strong guarantees mainly in safe/restricted subsets; unsafe code and C/C++ interfaces can reintroduce memory-safety bugs.
Sanitizers	△	△	×	High	✓	Object-level	Detection-oriented; coverage is limited to instrumented code and executed paths; high overhead for production real-time systems.
Metadata-based instrumentation	✓	✓	✓	High	✓	Pointer/object-level	Strong protection under recompilation, but requires pervasive metadata propagation and incurs significant runtime and memory overhead.
Dynamic binary instrumentation	△	△	×	High	✓	Object-level	Does not require source recompilation, but is testing/debugging oriented and limited to executed paths.
Managed runtimes / GC	△	✓	△	Med–High	×	Object-level	Bounds/type safety depend on the language runtime; garbage collection can introduce pause-time and timing-predictability concerns.
Static analysis / fuzzing	△	△	×	None–High	✓	N/A	Diagnostic/testing techniques; false positives/negatives; no global guarantees beyond analyzed or exercised paths.
Formal methods	✓	✓	✓	None	△	Program/system-level	Strong proof-based guarantees, but require specifications, annotations, verification expertise, and often scale poorly to large evolving systems.
Memory tagging	△	△	×	Low	✓	Granule-level	Probabilistic detection; tag collisions and coarse granularity may allow some violations to go undetected.
Pointer authentication	×	×	×	Low	✓	Pointer-level	Protects pointer integrity, but does not inherently enforce object bounds or lifetime validity.
Bounds/capability-based systems	✓	△	✓	Low–Med	△	Pointer/object-level	Provides deterministic hardware-checked metadata, such as bounds, permissions, or capability-like pointer authority; usually requires ISA/compiler/runtime changes and may not fully solve temporal safety
Hardening	△	△	×	Low–High	✓	Control/data-flow	Limits exploitability rather than eliminating memory corruption; precision, entropy, and bypass resistance vary
CHERI	✓	△	✓	Low–Med	△	Object-level	Strong spatial protection through architectural capabilities; compatible with C/C++ migration, but requires recompilation and adaptation of some legacy code patterns.

Table 8: Comparison of memory-safety techniques and positioning of CHERI. SMS: spatial memory safety; TMS: temporal memory safety; DET: deterministic guarantees; RO: runtime overhead; C/C++ Compat.: compatibility with existing C/C++ codebases. ✓: strong support, △: partial or limited support, ×: not supported.

checks, while GC [205, 206] mechanisms automatically manage memory by inserting barriers to trace memory accesses, track object lifetimes, and reclaim memory. Although these techniques can be highly effective, they typically incur nontrivial runtime overhead, affect execution timing, and often fail to provide comprehensive coverage. For example, GC must periodically suspend the application to synchronize and perform global memory management. These limitations are particularly critical in embedded and real-time systems, where predictability, tight worst-case execution time, and stringent certification requirements severely restrict the applicability of heavyweight runtime instrumentation. Static analysis tools [27, 28, 237] form a complementary line of defense and enable early bug detection without runtime cost, but often suffer from false positives and limited precision. Coverage-guided fuzzers [242–244] explore various execution paths by generating and mutating inputs, while runtime instrumentation frameworks, such as compiler-based sanitizers (e.g., ASan [45]) or binary instrumentation tools such as Valgrind [239], detect spatial, temporal, and undefined behavior violations along those paths. This combination has proven highly effective for vulnerability discovery and regression testing. However, its correctness guarantees are restricted to the concrete execution paths that are actually exercised, and the absence of reported violations cannot be interpreted as a proof of global memory safety. Furthermore, dynamic instrumentation incurs nontrivial runtime overhead and may introduce nondeterministic behavior, thereby constraining its applicability in production environments, particularly in performance-critical or real-time systems. Consequently, fuzzing, static, and dynamic analysis should be regarded as highly effective diagnostic and debugging techniques, but they do not provide deterministic guarantees of memory safety. Formal methods and verified toolchains complement defensive mechanisms by providing mathematical guarantees that implementations comply with their formal specifications. Verified compilers, for example, ensure that optimizations preserve the semantics of the source program and do not introduce new memory-safety violations during code generation. Deductive verification frameworks such as VCC [231] and VeriFast [230] enable developers to demonstrate memory safety and functional properties directly at the source level. However, such approaches typically require extensive manual annotations and precise specifications, and their scalability remains limited for large or evolving codebases. Hardware-assisted mechanisms such as memory tagging (e.g., Arm MTE [33, 220]) provide low-overhead runtime detection of memory-safety violations by associating lightweight metadata with memory regions and pointers. However, these approaches remain fundamentally probabilistic as their effectiveness depends on the entropy of the tag and the resistance to collisions. Pointer authentication mechanisms [226] protect pointer integrity by cryptographically binding pointer values to secret keys, preventing unauthorized substitution or reuse. While effective in detecting pointer corruption, pointer authentication does not intrinsically enforce memory bounds or object lifetime constraints. Extensions such as PAC-based bounds schemes (e.g., AOS [225]) require additional hardware structures to associate authenticated pointers with bounds metadata. Stack and heap hardening mechanisms, ASLR, memory-permission enforcement, and integrity-based defenses collectively aim to limit exploitability even when memory vulnerabilities are present. Stack canaries [257–259], shadow stacks [260, 261], and stack-pointer authentication protect control-flow data [121, 262],

while hardened allocators [266–269] use randomization, red-zones, quarantine, and type-based reuse restrictions to mitigate spatial and temporal exploits. ASLR [277] and memory-permission techniques [287, 288] further constrain attacker capabilities by introducing unpredictability and restricting code injection. Integrity-oriented defense mechanisms, such as CFI [290], CPI [299], and DFI [302], enforce structural constraints on control-flow transfers and memory writes to prevent control-flow hijacking and unauthorized manipulation of security-critical data. Although these mitigation techniques have demonstrated substantial practical effectiveness, they do not eliminate the underlying memory-corruption primitive. Randomization-based defenses are inherently probabilistic and can be compromised by entropy reduction, information disclosure vulnerabilities, or side-channel leakage. CFI mechanisms constrain indirect control transfers but cannot preclude semantically valid data-oriented corruption that occurs along otherwise legitimate control-flow paths. Pointer-integrity schemes safeguard particular classes of pointers, yet do not, by themselves, enforce object-bound constraints. DFI mechanisms restrict unauthorized memory writes, but typically incur substantial performance overhead and rely on conservative static analyses.

Positioning of CHERI in the Memory-Safety Landscape. Table 8 positions CHERI within the broader landscape of memory-safety techniques. In contrast to language-level safety mechanisms, software instrumentation, integrity-enforcement schemes, and probabilistic hardening techniques, CHERI elevates memory safety to a first-class architectural property. Instead of relying on dynamic checks, randomization, or structural constraints layered on top of conventional pointers, CHERI replaces raw pointers with capabilities that encode the bounds, permissions, and provenance directly in hardware. These capabilities are enforced at the ISA-level, ensuring that spatial memory violations and unauthorized accesses are architecturally precluded. This approach fundamentally redefines the memory-safety model. Whereas sanitizers detect violations only along executed paths, CFI constrains control-flow transfers, CPI protects selected classes of pointers, and DFI restricts write targets, CHERI enforces object-granular authority on every memory dereference. Consequently, traditional exploitation primitives, such as out-of-bounds buffer accesses and stack pivoting, cannot be employed to obtain unrestricted memory access, because capability bounds and permissions are enforced in hardware and validated on every capability use. Crucially, CHERI is not conceived as a replacement for complementary techniques, but rather as an architectural substrate upon which such mechanisms can be deployed more robustly. For instance, memory-tagging schemes can be implemented on top of CHERI to enable lightweight capability revocation. Nevertheless, the design of efficient and general-purpose temporal-safety guarantees based on capability revocation remains an open research challenge. In addition, CHERI complements formal methods and verified toolchains by providing a hardware foundation that preserves end-to-end safety properties from ISA-level semantics through system software, while remaining compatible with legacy C/C++ code and enabling incremental deployment without pervasive instrumentation or heavy-weight runtime checks. From a verification point of view, CHERI changes the role of static analysis, sanitization frameworks, and fuzz testing: these techniques remain valuable for detecting logical defects, unsafe-code misuse, temporal-safety issues, and higher-level semantic violations, while many spatial memory-safety violations are constrained

by architectural enforcement. From a systems-theoretic point of view, CHERI constitutes a fundamental architectural shift: memory safety is relocated from probabilistic mitigation mechanisms and integrity enforcement to deterministic hardware-mediated authority semantics. Instead of attempting to constrain corrupted states after their manifestation, CHERI restricts the expressive power of memory accesses such that broad classes of memory corruption are rendered inexpressible within the architectural model itself.

3.3.2 CHERI vs. Other Compartmentalization and Isolation Technologies

Technique	IB	GCM	IDP	KGP
Monolithic OS / RTOS	Processes, tasks, or kernel/user separation.	Coarse-grained; low overhead, but large trusted kernel/runtime.	**No / Limited.** Components may share the same protection domain.	Large TCB; memory-unsafe kernels, drivers, libraries, or RTOS tasks can corrupt other components within the same domain.
Microkernels	Address spaces and kernel-mediated capabilities.	Medium granularity; lower kernel TCB, but isolation usually requires address-space separation and IPC.	**Limited.** Strong across address spaces, but user-space components remain vulnerable internally.	Fine-grained decomposition requires many address spaces and frequent IPC, increasing engineering complexity.
Virtualization-Based Isolation	Static partitions, guests, or VMs, often realized by SPHs.	Coarse-grained; strong isolation through static resource allocation and minimal sharing.	**No.** Components inside each partition or guest are usually mutually trusted.	Well suited for mixed-criticality consolidation and certification, but isolation remains at partition/VM granularity and does not provide object-level protection within guests or hypervisor subsystems.
MPU-based Isolation	Static memory regions.	Very coarse-grained; low and predictable overhead, but limited number of regions.	**Limited.** Protects regions, not individual objects.	Poor scalability and rigid alignment/region constraints may force unrelated objects into the same protection region.
TEEs / Enclaves	Worlds, enclaves, or trusted execution domains.	Coarse-to-medium granularity; nontrivial transition and management costs.	**No.** Code inside the trusted domain remains vulnerable internally.	Protects trusted domains from external software, but does not provide fine-grained isolation inside the trusted domain itself.
Confidential VMs	VMs protected from a privileged host or hypervisor.	Coarse-grained; VM-level isolation with memory-encryption/attestation costs.	**No.** Protection stops at the VM boundary.	Confidentiality and integrity are improved at the VM boundary, but intra-VM memory corruption remains unaddressed.
Microarch. Isolation	Shared hardware resources: caches, TLBs, predictors, DRAM, and interconnects.	Resource-level; partitioning, flushing, randomization, or bandwidth regulation.	**No.** Does not enforce software memory safety.	Reduces timing interference and side channels, but does not prevent memory corruption inside architectural domains.
CHERI	Intra-address-space capabilities.	Fine-grained object-level authority with hardware-enforced checks.	**Yes.** Enforces bounds, permissions, and provenance.	Provides deterministic spatial safety and fine-grained compartmentalization, but its use in MCS remains underexplored, particularly with virtualization and static partitioning.

Table 9: Comparison of compartmentalization and isolation techniques for MCS from a CHERI-oriented gap perspective. The table analyzes each technique in 4 dimensions: (i) isolation boundary (IB); (ii) granularity/cost model (GCM); (iii) intra-domain protection (IDP); and (iv) key gap/positioning (KGP).

Table 9 summarizes the design space of existing compartmentalization and isolation techniques across key dimensions, including isolation level, granularity, trusted computing base size, performance overhead, real-time suitability, and intra-domain memory safety. The following discussion interprets these dimensions and highlights the fundamental limitations that motivate the use of CHERI. Microkernels [51,

53, 306] reduce kernel complexity by pushing services (e.g., drivers, filesystems) to user-space applications, enforcing isolation primarily through address-space separation and capability-mediated access to kernel objects. Similarly, separation kernels and safety-oriented hypervisors [118, 124, 312, 315] provide strong spatial partitioning, minimized TCB, and controlled communication channels aligned with safety certification goals. TEEs and CC (e.g., TrustZone [58], enclaves [368, 376, 377], and confidential VMs [63, 64]) further strengthen isolation under threat models that include an untrusted OS or hypervisor. Across these designs, isolation boundaries are typically defined at coarse granularity (processes, partitions, tasks, VMs, worlds, and enclaves), and are enforced via privilege separation and page/region-based protection, sometimes augmented by dedicated system-wide security hardware isolation extensions (e.g., TrustZone, Arm CCA, RISC-V CoVE, and WorldGuard) [58, 63–65]. Although effective for domain-level compartmentalization, this model does not naturally support fine-grained decomposition within a single address space. To isolate subsystems under this paradigm, system designers must introduce additional protection domains, typically using distinct processes or VMs, which in turn require frequent address-space transitions, increased memory footprint (memory is partitioned into pages) and IPC-mediated communication. These mechanisms impose nontrivial performance overheads, increase architectural and implementation complexity, and complicate timing analysis in real-time and MCS. Consequently, fine-grained isolation is often eschewed in practice due to its engineering burden and runtime costs. This limitation is particularly pronounced in deeply embedded safety-critical systems. Such systems often use monolithic RTOSs that provide only coarse-grained isolation guarantees, or they execute entire software stacks within statically partitioned VMs. On platforms that lack virtual memory support, isolation is commonly realized through MPUs. Although MPUs offer low-overhead, region-based and largely deterministic protection with predictable execution, they typically expose only a small number of protection entries and impose rigid alignment constraints. These restrictions often force the aggregation of semantically unrelated objects within the same protection region, thereby reducing the effective granularity of the compartment and amplifying the consequences of memory-safety violations. Virtualization-based consolidation [15, 118, 132] exhibits analogous structural characteristics. Although it provides strong isolation at the granularity of the entire VM, the decomposition of software within each guest continues to rely on conventional mechanisms, inter-partition communication is typically realized through explicitly configured shared-memory regions or buffer copying, and device multiplexing may rely on paravirtualization mechanisms (e.g., VirtIO [137]), which are frequently implemented in dedicated service domains. Even for formally verified kernels such as seL4 [51], machine-checked guarantees pertain primarily to kernel-level functional correctness and information-flow enforcement. User-space components, frequently implemented in memory-unsafe languages, remain susceptible to intra-domain memory corruption and depend on kernel mediation to interact with other subsystems. In summary, the prevailing isolation techniques are fundamentally organized around address spaces and hardware-enforced privilege levels. Although they provide strong separation between protection domains, they lack an efficient, scalable mechanism for fine-grained, object-level isolation within a single domain. As systems scale and consolidate increasingly heterogeneous workloads, TCBs tend to increase in size and complexity, thereby elevating both the probability and the potential impact of intra-domain

vulnerabilities, despite the presence of robust, structurally enforced isolation mechanisms. In parallel, microarchitectural and memory-system isolation techniques (e.g., cache partitioning [327, 353], TLB isolation [337, 338], branch-predictor mitigations [340, 358], and DRAM bandwidth regulation [345–349]) complement architectural isolation by attenuating timing interference and side-channels that originate from contention on shared hardware resources. These mechanisms are essential for ensuring freedom-from-interference execution and for limiting the capacity of building covert channels in multicore systems. However, they do not address the fundamental problem of software-induced memory corruption. Instead, they primarily restrict cross-domain interference and information leakage among execution compartments that are already isolated at the architectural level.

Positioning of CHERI in the Isolation Landscape. Table 9 positions CHERI within the broader landscape of compartmentalization and isolation technologies. As shown, existing approaches primarily enforce isolation at coarse granularity and do not provide protection against intra-domain memory corruption. CHERI occupies a fundamentally distinct position within the isolation design space by enabling fine-grained, hardware-enforced compartmentalization *within* a single address space. CHERI offers a principled architectural substrate for the systematic (re)design of microkernels, OS kernels, and hypervisors, by permitting subsystems such as libraries, device drivers, and system services to execute with strictly minimized privileges while co-existing in a shared virtual address space, thereby obviating the complexity and overhead traditionally associated with managing multiple address spaces. From the perspective of a hypervisor, CHERI introduces an orthogonal protection mechanism that augments traditional virtualization-based partitioning. Conventional hypervisors isolate entire guest software stacks, but typically rely on an assumption of mutual trust among components within an individual guest. Achieving fine-grained intra-guest decomposition, such as isolating device drivers, protocol stacks, or runtime services, usually requires additional address spaces. With CHERI, guest subsystems can instead be compartmentalized at object-level granularity within a single guest address space, thereby diminishing the reliance on heavy-weight domain boundaries. Hypervisor services, including paravirtualized backend drivers (e.g., VirtIO backends), which are commonly placed in dedicated VM instances to restrict fault propagation, could alternatively be realized as mutually isolated capability-based compartments running within the hypervisor itself or within a consolidated execution context. This design maintains the principle of least privilege while reducing the overhead associated with inter-domain crossings. However, despite this promising design space, the interplay between CHERI's architectural protection model and conventional virtualization architectures remains, to a large extent, inadequately characterized and empirically underexplored. Although prior work has demonstrated the feasibility of OSs and microkernels based on CHERI [87, 99, 387], the systematic exploration of CHERI-augmented hypervisor architectures and guest decomposition strategies in mixed-criticality workloads remains an open research direction. In microkernel contexts, systems such as CheriOS [387] illustrate how a pure-capability architecture can support a unikernel-style model in which services co-exist within a unified address space yet remain strongly isolated through capabilities. This eliminates reliance on traditional ring-based privilege separation and reduces the semantic gap between

software abstractions and hardware protection mechanisms. Importantly, CHERI may also constitute a scalable alternative to MPU-based isolation in MMU-less virtualization environments that demand strongly deterministic and predictable execution. In this model, CHERI provides a promising architectural foundation for MMU-less virtualization schemes with substantially improved scalability, enabling in principle an unbounded number of security domains, in contrast to conventional MPU-based approaches. Nevertheless, CHERI can be composed with other system-level isolation mechanisms. For instance, microkernels such as seL4 [51] can retain formally verified structural isolation, with CHERI extending protection into user space by enforcing deterministic spatial safety at the hardware level. Despite the strong architectural isolation provided by CHERI, it does not eliminate all classes of vulnerabilities. Defects affecting logical correctness, microarchitectural side channels, and timing-based information leaks remain outside the scope of its threat model. Therefore, CHERI-based systems continue to require cache, TLB and memory-system isolation mechanisms to provide strong temporal guarantees in single-core and multicore environments. Furthermore, implementing scalable capability revocation and ensuring controlled temporal safety across security domains remains a technically challenging and open research issue in the literature.

4 Concluding Remarks

This chapter concludes the dissertation by summarizing its main contributions and distilling the key conclusions drawn from the preceding chapters. In particular, it revisits the research questions formulated in Section 1.2.2 and summarizes how they are addressed by the results of this work. The chapter also reflects on the broader implications of these findings for the design of MCSs and outlines promising directions for future research (Section 4.2).

4.1 Conclusions

This dissertation advances the state of the art in capability-based architectures as a foundation for enhancing memory safety and isolation in contemporary MCS. Motivated by the increasing consolidation of heterogeneous workloads onto shared hardware platforms, from safety-critical real-time applications to general-purpose software, this work investigates the feasibility and implications of adopting CHERI technology in RISC-V-based virtualization stacks for MCS. In contrast to prior research that typically considers capability architectures or virtualization in isolation, this dissertation systematically examines their interaction across the full system stack. By combining ecosystem analysis, microarchitectural design and evaluation, hypervisor integration, and system-level experimentation, it provides a comprehensive assessment of the feasibility, overheads, and security implications of CHERI-enabled virtualization on RISC-V platforms. The results show that CHERI can be incorporated into contemporary hypervisor-based designs with modest performance and complexity overhead, while enabling fine-grained intra-domain isolation that is difficult to achieve with conventional mechanisms based solely on MMUs or MPUs. The following paragraphs summarize the main contributions of this thesis and explicitly answer the research questions formulated in Section 1.2.2.

CHERI Ecosystem Maturity.

In addressing **RQ1**, this dissertation presented a detailed and structured survey of the CHERI ecosystem, consolidating developments across hardware, software, tooling, and standardization efforts. The survey shows that CHERI has evolved from a primarily academic research concept to a rapidly maturing technology with growing industrial relevance and institutional backing. At the hardware level, CHERI has

seen significant adoption in both microcontroller-class and application-class processors. In particular, RISC-V has emerged as the dominant open architecture for CHERI experimentation and deployment, with implementations such as CHERIoT [87] targeting deeply embedded systems and application-class processors such as CHERI-Flute [165] and CHERI-Toooba [166] enabling the deployment of full-featured OSs. In parallel, the ongoing ratification of the RISC-V CHERI extension (`Zcheri`) represents a critical step toward standardization of the CHERI extension and ecosystem development. Research platforms, including Arm Morello [83] and Sonata [86], have further accelerated adoption by providing accessible development and experimentation environments. From a software perspective, the support for CHERI has expanded throughout the system stack, including OSs (CheriBSD [99] and CHERI Linux [101]), RTOSs (e.g., CHERIoT RTOS [388] and CHERI-Zephyr [103]), and runtime environments. Tooling support has also matured substantially, with emulators (e.g., QEMU [389]), verification frameworks (e.g., TestRIG [390]), and compiler infrastructure enabling reproducible evaluation and validation. Finally, a significant body of recent literature focuses on capability revocation and lifecycle management, a critical challenge for the widespread adoption of CHERI. Collectively, this survey establishes a clear snapshot of the current state of CHERI technology and provides a structured reference to understand its readiness, limitations, and future research directions.

CVA6 Hypervisor Extension and CHERI Support.

To evaluate the feasibility and costs of integrating the RISC-V hypervisor extension with CHERI and to address **RQ2** and **RQ3**, this dissertation presented a comprehensive hardware and system-level evaluation based on the open-source CVA6 core, extended to support both the RISC-V hypervisor and CHERI hardware extensions. The work was divided into two main parts: (i) implementation of the hypervisor hardware extension and associated microarchitectural optimizations on the CVA6 core and (ii) implementation of the CHERI RISC-V ISA extension atop the CVA6 core with hypervisor extension support.

In addressing **RQ2**, we added hardware virtualization support to the RISC-V CVA6 processor and introduced optimizations to critical microarchitectural structures affected by virtualization, with the goal of reducing overheads from nested MMU translation and timer management. First, to mitigate slowdowns caused by nested page-table walks, we proposed and implemented a dedicated second-stage GTLB to accelerate PTW processing on TLB misses. Second, we introduced a unified L2 TLB that supports multiple page sizes (4 KiB and 2 MiB) with configurable capacity. In addition, we implemented the standard RISC-V timer extension Sstc, which provides timer interfaces at both hypervisor and guest levels, eliminating the need for timer emulation in HS-mode and VS-mode. Furthermore, we conducted a comprehensive design-space exploration on an FPGA platform to measure functional performance (in execution cycles) and hardware cost. This exploration covered more than 23 design configurations, including L1 TLB size, enablement and size of GTLB, L2 TLB capacity and associativity, separate entries per page size, and activation of Sstc extension. Based on these results, we selected a subset of configurations and performed a detailed PPA analysis using a 22 nm FDX technology node. The analysis revealed several design points that illustrate trade-offs between performance, power consumption, and silicon area. We achieved up to

19% performance improvement at the cost of a 10% area increase, and selected an optimal configuration that provides an average speedup of 12.5% with minimal overhead in area (0.78%) and power (0.33%). This implementation was upstreamed to the official CVA6 repository and, at the time of writing, verification efforts were ongoing with partners in the OpenHW Group.

In addressing **RQ3**, we implemented the CHERI version 9 extension on top of the CVA6 core. As part of this work, we implemented a dedicated tag controller to manage capability tags in memory and ported the standard CHERI CapLib Register-Transfer Level (RTL) API from Bluespec to SystemVerilog. This library provides an abstraction for capability management and is compliant with the specification of the Concentrate 128-bit capability format. We also adapted the core to support verification frameworks such as TestRIG by implementing RVFI-DII, which supports instruction tracing and injection into the pipeline. Finally, we performed a comprehensive FPGA-based functional evaluation of the extension in a single-core CVA6-based SoC using three representative benchmark suites (MiBench, San Diego Vision, and CoreMark/MHz) to quantify performance overhead in execution cycles. The results showed that CHERI introduces slowdowns ranging from 1.1× to 3.2× in the worst case compared to a FreeBSD-based stack, together with FPGA resource utilization overheads of approximately 53.9% in Lookup Tables (LUTs) and 43.0% in registers compared to the design enabled by the baseline hypervisor. Most of the overhead is concentrated in the system time, due to capabilityrevocation, whose costs can be reduced using modern techniques [391]. In collaboration with the University of Cambridge, the design was updated to comply with the forthcoming standard RISC-V extension CHERI (`Zcheri`) and enhanced with microarchitectural optimizations (e.g., checking Program Counter Capability (PCC) bounds later in the pipeline and a state-of-the-art tagged memory controller) and a preliminary area analysis.

Bao and CHERI Integration.

In addressing **RQ4**, this dissertation integrated CHERI into the SPH Bao, resulting in the first open-source CHERI-aware hypervisor for RISC-V.

Feasibility. We successfully compiled and executed Bao in pure-capability mode on a CVA6H-CHERI-based platform, booting and hosting both a CHERI-aware guest (CheriBSD) and legacy guests (bare-metal applications). This shows that CHERI can be composed with the RISC-V hypervisor extension without architectural incompatibilities, preserving the static-partitioning model of Bao while adding capability-based memory protection.

Engineering costs. In addition to the hardware resources at the platform level required to support CHERI, enabling it in Bao introduces moderate but measurable overheads. The static binary size of Bao increased substantially (about `.text` +30%, `.bss` +59%, `.data` +26%), largely due to capability-aware instruction sequences and expanded data structures. The runtime memory usage to save the architectural state context increased by approximately 280 B per-vCPU. Boot-time measurements showed a systematic overhead of 20.1%, with average boot latency increasing from 35.4 ms (baseline Bao) to 42.5 ms (Bao-CHERI). This overhead is primarily due to CHERI-specific initialization, in particular processing the

`__cap_relocs` metadata required to materialize bounded global and static capabilities. Interrupt latency experiments further quantify the runtime impact: average interrupt latency increased from 89.8 µs to 101.9 µs (+13.4%), while worst-case latency increased by approximately 13.0%. The application-level benchmarking using MiBench automotive workloads showed an additional average slowdown of about 10% in hosted environments compared to the vanilla Bao. When combined with virtualization overheads, worst-case slowdowns exceeded 40% for short-running, system-intensive benchmarks; further microarchitectural optimizations are expected to reduce this cost.

Security and isolation. Bao-CHERI enables isolation models that are difficult to achieve with conventional MMU/MPU-based hypervisors alone. Running the hypervisor in pure-capability mode enforces fine-grained, object-level memory safety within the hypervisor itself: critical structures (CPU state, VM metadata, trapframes, and stacks) are protected with bounded capabilities, reducing ambient authority and limiting the impact of memory-safety vulnerabilities. CHERI also enables principled intra-hypervisor compartmentalization and capability-mediated sharing, providing a foundation for safer VirtIO backends and predictable isolation even in MMU-less virtualization scenarios. In general, the Bao-CHERI integration shows that CHERI can be deployed in statically partitioned hypervisor environments, providing strong security benefits at a quantifiable and manageable cost and supporting the future of embedded MCS.

Impact and Adoption. The outcomes of this work extend beyond the proposed architectural and microarchitectural contributions, having a direct impact on both the open-source and standardization ecosystems. The implementation of the RISC-V hypervisor extension and its associated optimizations has been upstreamed to the official CVA6 repository, maintained by the OpenHW Group, with verification efforts ongoing in collaboration with OpenHW partners. As a result, these contributions are now part of an actively maintained, industrial-grade open-source core and are being used in several European research projects and industrial initiatives, including REBECCA, ISOLDE, and RIGOLETTO, as well as by industrial partners such as Thales and research platforms such as Cheshire [184] and Carfield [185].

In parallel, the CHERI-enabled CVA6 platform has contributed to ongoing standardization efforts within the RISC-V ecosystem. In particular, the implementation has been used as a reference point for validating aspects of the emerging `Zcheri` specification, supporting the activities of the RISC-V CHERI working group. Furthermore, industrial stakeholders, including Capabilities Limited, have adopted and extended the design towards commercialization as a capability-enabled processor IP. These developments highlight the relevance of this work not only as a research prototype, but also as a foundation for future industrial-grade CHERI-aware RISC-V systems.

4.2 Future Work

This thesis provided the first open-source reference stack combining CHERI with RISC-V virtualization support. Future work spans multiple dimensions, including improvements to address translation and memory

subsystems, architectural and microarchitectural enhancements for capability-aware execution, deeper integration of CHERI with virtualization and I/O isolation, and comprehensive evaluation in multicore and system-level environments. Collectively, these directions aim to enable a broader end-to-end assessment of capability-based protection in MCS, including performance, scalability, and interference behavior.

4.2.1 Memory Subsystem Improvements

From a microarchitectural perspective, efficient address translation remains a critical challenge in virtualized systems, particularly when combined with capability-based protection mechanisms such as CHERI. Our evaluation on CVA6 highlights that the memory subsystem, and specifically the MMU, can become a dominant source of performance degradation, motivating further research into more scalable translation mechanisms. One promising direction is the use of TLB coalescing techniques, whereby page-table entries encode contiguous memory regions using Naturally Aligned Power-of-Two (NAPOT)-style mappings (e.g., via the proposed RISC-V `Svnapot` extension). This approach allows a single TLB entry to cover multiple pages, reducing TLB pressure and lowering the frequency of page-table walks. Although demonstrated in the context of CVA6, this optimization is broadly applicable to virtualized architectures that rely on multi-level address translation. A complementary research direction concerns the optimization of page-table-walk mechanisms. Although our implementation introduces a dedicated second-stage TLB to accelerate PTW, misses in this structure still require multiple accesses to the memory hierarchy to retrieve the page tables, increasing latency and inducing cache pollution and interconnect traffic. A promising approach is the integration of a small Page Table Entry (PTE) cache with the PTW, which allows recently accessed page-table entries to be served locally. This design could reduce translation latency and alleviate pressure on shared memory resources. Finally, the current evaluation targets a single-core FPGA configuration without an LLC. Although these results provide useful insights into translation behavior, evaluating these optimizations in multicore environments remains essential to fully understand scalability and system-level interactions. This aspect is further discussed in Section 4.2.4.

4.2.2 Architectural and Microarchitectural Directions for CHERI

This work establishes a CHERI-enabled RISC-V platform that serves as a concrete foundation to explore the architectural and system-level implications of capability-based protection. Building on this platform, several architectural and microarchitectural directions can be explored to further improve performance, scalability, and security.

Microarchitectural Optimizations. Although the current design performs an early PCC prediction, deferring bounds and permission checks to later pipeline stages could reduce complexity in the `frontend`, leveraging the in-order execution model of CVA6. This approach may improve timing closure and reduce pressure on the critical path. Another promising direction is to extend the Tag Controller with a hierarchical tag cache to reduce tag-access latency and alleviate pressure on the memory hierarchy. Furthermore,

the current tag-inspection mechanism tightly couples tag and data fetches, which may be suboptimal for tag-only operations such as revocation. Supporting direct multi-tag inspection from main memory, without fetching the associated data payload, could significantly improve the efficiency of such operations.

Temporal Memory Safety. Although CHERI provides strong spatial memory safety guarantees, temporal safety (e.g., use-after-free) remains a challenge. Proposals such as Cornucopia [392, 393] and CHERIoT [87] explore hardware-software co-design approaches for capability revocation and lifetime tracking. Integrating such mechanisms would enable stronger end-to-end memory safety guarantees, particularly for long-running and safety-critical systems. However, an important open question is whether these approaches are compatible with mixed-criticality requirements. In such systems, heterogeneous workloads with different criticality levels must coexist under strict timing guarantees and bounded interference. Temporal safety mechanisms typically introduce additional runtime activity (e.g., revocation scans, metadata updates, or deferred reclamation), which may lead to timing variability, increased contention in shared resources, and reduced predictability. This behavior conflicts with the determinism required in real-time and safety-critical environments. As a result, the applicability of temporal memory-safety mechanisms in MCS remains uncertain. Future work should therefore investigate revocation mechanisms specifically designed for these environments, focusing on deterministic, bounded-latency, and interference-aware approaches. Such mechanisms must ensure that temporal safety can be achieved without compromising predictability, analyzability, and freedom-from-interference guarantees.

Capability Sharing Across Protection Domains. Another promising direction is the efficient sharing of capabilities across protection domains, such as between virtual machines or between guest and hypervisor. In particular, physical capabilities [387, 394], which bypass virtual memory translation and directly reference physical memory, could enable zero-copy data sharing without relying on bounce buffers or page remapping. This approach could be particularly beneficial for high-performance I/O paths (e.g., VirtIO backends), although it would require further architectural support and analysis of the isolation guarantees.

Side-Channel Resistance and Isolation. Although CHERI enforces strong architectural isolation, it does not inherently prevent microarchitectural side-channel attacks, particularly those exploiting speculative execution and shared resources. Prior work [395] has shown that CHERI can mitigate certain classes of attacks (e.g., Meltdown), but remains vulnerable to others (e.g., Spectre-style attacks and capability speculation). Moreover, the increased size of capabilities and associated metadata may amplify pressure on caches and memory systems, potentially exacerbating timing-based leakage. An important open question is how capability-based protection interacts with shared microarchitectural resources such as caches, TLBs, and interconnects, which remain outside the direct protection scope of CHERI. Future work should therefore investigate microarchitectural mechanisms and system-level techniques, including cache partitioning, temporal isolation, and speculation control, to provide stronger and analyzable guarantees against timing and contention-based side channels in CHERI-based systems.

4.2.3 Virtualization, I/O Isolation, and Compartmentalization

A central direction for future work is the deeper integration of CHERI into full virtualization stacks. In MMU-based systems, virtualization traditionally enforces isolation at page granularity. Future work should assess how CHERI can complement, and in some scenarios reduce reliance on, page-based mechanisms to improve determinism by enabling (i) capability-mediated inter-process communication, (ii) fine-grained and revocable shared memory between virtual machines and the hypervisor, and (iii) intra-VM and hypervisor-level compartmentalization of services. A concrete example is VirtIO in Bao, where backend services are executed in a dedicated backend VM and communicate with the frontend VMs via shared-memory regions. An important direction is to move selected backend functionality into compartmentalized hypervisor tasks and use CHERI capabilities to share memory safely, for example by granting each frontend only bounded capabilities to its data structures. This design could reduce the need for bounce buffers when copying data into statically shared memory regions.

Beyond CPU isolation, practical hypervisors require secure I/O multiplexing and device passthrough. This is typically achieved using an IOMMU to restrict DMA accesses. Exploring the interaction between CHERI-based isolation and IOMMU-enforced DMA protection is therefore essential to achieve end-to-end isolation across both computation and I/O paths. Recent work further demonstrates that CHERI compartments can be used to deprivilege low-level Graphics Processing Unit (GPU) drivers by restructuring driver components into user-space processes and fine-grained compartments [396]. This approach reduces the TCB while preserving performance, providing a concrete blueprint for applying CHERI-based compartmentalization to complex I/O subsystems that are traditionally fully privileged.

4.2.4 Performance, Multicore Evaluation, and PPA Analysis

The evaluation in this dissertation focused primarily on functional workloads and single-core configurations, which were sufficient to characterize the initial integration costs and functional behavior of the proposed stack. Future work should extend this analysis to multicore platforms and expand the evaluation by (i) experimenting with alternative hypervisors and configurations (e.g., multiple guests sharing a single-core), (ii) running additional commercial benchmark suites such as SPECint [397], and (iii) migrating to multi-core platforms to study scalability and interference effects. In parallel, we are conducting a PPA analysis of the CHERI-enabled CVA6 design using an industry-relevant technology node to quantify the practical cost of CHERI integration. We also plan to tape out the design on the AISaqr chip [108]. Based on the results presented in this dissertation and prior observations on virtualization extensions, we expect that, with careful design, the area and power overheads of CHERI-specific components can be kept modest.

Part II

Publications

List of Publications

Publications in this Thesis

[111] **B. Sá** et al. "Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology". [Accepted at *ACM Computing Surveys*]

[112] **B. Sá** et al. "CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration". In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023)

[113] **B. Sá** et al. "CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation". [Submitted to *IEEE Transactions on Computers*]

[398] **B. Sá** and S. Pinto. "Bao-CHERI: A RISC-V Open-Source Hypervisor with CHERI Support". [Under Submission to *IEEE Access*]

Other Contributions

[110] **B. Sá** et al. "Holistic RISC-V Virtualization: CVA6-based SoC". In *20th ACM International Conference on Computing Frontiers* (2023).

[399] **B. Sá** et al. "RISC-V Virtualization: A Case Study on the CVA6". Poster and presentation at *RISC-V Summit* (2023).

[108] L. Valente, **B. Sá** et al. "Shaheen: An Open, Secure, and Scalable RV64 SoC for Autonomous Nano-UAVs". In *IEEE Hot Chips 35 Symposium (HCS)* (2023).

[400] L. Valente, A. Nadalini, A. Veeran, M. Sinigaglia, **B. Sá**, et al. "A Heterogeneous RISC-V Based SoC for Secure Nano-UAV Navigation". In *Transactions on Circuits and Systems I: Regular Papers* (2024).

[109] F. Costa, M. Rodríguez, **B. Sá**, and S. Pinto. "Interrupting the Status Quo: A First Glance at the RISC-V Advanced Interrupt Architecture (AIA)". In *IEEE Access* (2024).

[399] **B. Sá** et al. "CHERI RISC-V: A Case Study on the CVA6". Poster and presentation at *RISC-V Summit* (2024).

[401] **B. Sá** et al. "Bao-CHERI: A Pure-Capability RISC-V Hypervisor". Poster at *RISC-V Summit* (2026).

5

Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology

This chapter was published as:

Bruno Sá, Donato Ferraro, Everton de Matos, Rafail Psiakis, Andrea Marongiu, Andrea Bastoni, and Sandro Pinto "Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology". In *ACM Computing Surveys*.

Abstract

As next-generation systems become increasingly complex and interconnected, they face the burden of new and numerous critical security challenges. CHERI has emerged as a promising technology to mitigate memory safety vulnerabilities, one of the major security threats in current computing systems. CHERI introduces a hardware-enforced, fine-grained memory protection model that integrates a fat-pointer representation, combining memory bounds and permissions to ensure safer memory access and mitigate common memory-related exploits. Despite its potential, information about CHERI evolution, design principles, and applications is scattered among multiple sources, making it difficult to fully grasp and correlate its features and applications. This paper aims to fill this gap by providing a detailed survey of CHERI technology, covering its historical evolution, key concepts, and current hardware and software implementations. We explore the ongoing challenges of its adoption in industry, the research efforts driving its growth as well as speculate and discuss future research directions.

5.1 Introduction

As software systems become increasingly complex and interconnected, the need for robust system security has become paramount. Memory safety issues are among the prevailing and most dangerous vulnerabilities that affect modern systems [402]. These vulnerabilities allow an attacker to exploit bugs in the memory management, leading to security breaches such as remote code execution, unauthorized access to sensitive data, and even the compromise of the entire system. The severity of these risks is amplified by the fact that memory safety vulnerabilities are often difficult to detect and mitigate, particularly in large, legacy codebases. This challenge is critical in languages like C and C++, which, despite being widely used in safety-critical applications, lack built-in memory safety mechanisms. These programming languages offer low-level control over memory allocation and management, trading high performance for poor security guarantees. As a result, many vulnerabilities, such as *buffer overflows* and *use-after-free* errors, are commonly found in software written in these languages [87].

As the number of cyber-attacks exploiting these vulnerabilities continues to rise, the issue of memory safety has reached the peak of attention in recent years. According to a research conducted by Microsoft [24], between 2006 and 2018, more than 70% of the security patches released were related to memory safety issues. Among these, spatial safety vulnerabilities, which involve accessing memory regions outside permitted bounds, were the most common category [24]. Memory safety issues have also been reported in Google Chromium [403] and in Apple OS [25], emphasizing the need for innovative solutions capable of addressing the evolving threats posed by increasingly sophisticated attacks.

In response to the growing dangers of memory safety vulnerabilities, academia and industry have explored a plethora of mitigation approaches. Traditional software-based solutions include techniques designed to identify, mitigate, or prevent vulnerabilities at different stages of the software development lifecycle. These techniques rely on either static analysis or dynamic analysis tools. Static analysis tools

(e.g., CppCheck [404]) analyze the program source code to detect potential vulnerabilities, such as buffer overflows, *null* pointer dereferences, or use-after-free bugs, and produce detailed reports [190, 405, 406]. However, these tools fall short when applied to large software codebases since they need to analyze all possible execution paths. Moreover, multiple tools applied to the same code can produce similar, but still different results, including possible false positives [407]. Other approaches aim at replacing memory-unsafe programming languages (e.g., C/C++) with newer, memory-safe programming languages. For instance, Rust's *ownership* and *borrowing* are unique features that enable memory safety guarantees at compile-time [38]. However, rewriting the entire software stack is not always possible [190, 408]. Additionally, especially for low-level software, Rust is not fully secure, as it allows the use of "unsafe" code, e.g., to interoperate with the hardware. Such unsafe constructs can bypass compile-time checks and are widely used in Rust environments [409]. Dynamic analysis tools, like AddressSanitizer (ASan), instrument code at compile time to introduce lightweight runtime checks, using mechanisms like *redzone* and *shadow memory*, to detect memory-related bugs during the execution of the application. Nevertheless, these tools introduce overhead in both performance and memory utilization [30]. Hardware-assisted mechanisms, such as Arm MTE [220], use embedded metadata (tag bits) to associate memory regions with pointers and to detect memory-safety violations at runtime via tag mismatches. While these mechanisms are relatively easy to integrate into existing systems without sacrificing compatibility and can provide broad coverage with modest performance overhead, they are inherently probabilistic. As a result, they remain vulnerable to tag collisions, where semantically unrelated objects receive identical tags, a condition that attackers can exploit to bypass protection.

An orthogonal approach to improve memory safety is the adoption of *capability*-based systems that enforce security by embedding access permissions directly into memory references. A fundamental property of these systems is *unforgeability*, which guarantees that valid capabilities cannot be arbitrarily created or modified by untrusted components, thereby protecting their integrity. Capability-based systems naturally enforce the *principle of least privilege* [143], i.e., every software component is granted only the minimum set of access rights to a resource to complete its functions and effectively mitigate the "confused deputy" problem [71]. Additionally, the *principle of intentional use* [163] ensures that these access rights are bounded within their intended scope. Capabilities can be implemented either in hardware or software. Software capabilities, as implemented in, e.g., seL4 [51], embed access permissions into the program logic, offering flexibility and architecture independence. In contrast, hardware capabilities, such as in IBM System/38 [74] and SAFE [157], integrate these controls at the architectural level, providing enhanced robustness and fine-grained control.

In this context, CHERI [37, 394], pioneered in 2014 by the University of Cambridge Computer Laboratory and the SRI International's Computer Science Laboratory, is gaining momentum as a promising approach to mitigate memory-related vulnerabilities. CHERI introduces a novel, hardware-enforced pointer representation referred to as "*capability*," i.e., an unforgeable fat-pointer [37] that combines the memory address with its metadata, such as bounds and permissions. Based on this metadata, CHERI enables hardware-enforced access controls that prevent unauthorized memory access, effectively mitigating

many common memory safety vulnerabilities like buffer overflows. CHERI offers improved security with minimal impact on performance [410], and fine-grained byte-level memory protection. This differs from other capability-based solutions, especially software-based ones, e.g., seL4 [51] and Fiasco.OC [52] as discussed in Section 5.2.2. CHERI not only mitigates buffer overflow attacks, but also contributes to temporal memory safety by supporting mechanisms that can detect or prevent use-after-free issues (detailed in Section 5.5.4.2). Moreover, CHERI enables system *compartmentalization*, i.e., the ability to securely isolate different system components into compartments. This feature prevents a faulty compartment from accessing others. In order to provide such protection, CHERI integrates specialized hardware with modifications across the entire system stack, including microarchitectural changes, compilers, operating systems, libraries, and applications, making it a deeply integrated and complete security technology.

After its first introduction by Woodruff et al. [37, 411], over the past decade, research and development in CHERI have spanned over multiple domains, reflecting its vast applicability and driving enhancements in both hardware and software security. At the hardware level, CHERI has evolved into different ISA extensions, with a focus on improving the security, performance, and footprint of the ISA. Furthermore, we have witnessed the rise of hardware platforms such as *Arm Morello* [83] and *Sonata* [86], which enabled the evaluation of real performance and facilitated further CHERI improvements. On the software side, considerable efforts have focused on porting compilers and operating systems, developing secure applications, and integrating CHERI with existing software stacks.

The CHERI technology has demonstrated potential across multiple use cases, including traditional computing environments, cloud infrastructure, and real-time embedded systems. The UK government's Industrial Strategy Challenge Fund (ISCF) [412] initiative has provided funding to accelerate CHERI's research and adoption, with programs such as DSbD [85], which drives collaborative projects between academia and industry. As a matter of fact, this initiative has facilitated the growth of industry support for CHERI, with notable contributions from organizations like Arm, through the *Arm Morello* [83] platform, and Microsoft's CHERIoT initiative [87]. Other companies such as Codasip [84] and Huawei [101] are exploring CHERI integration into RISC-V architectures. As a result, in November 2024, the *CHERI Alliance* [106] was established to promote the widespread adoption of CHERI and involves industry members like Codasip, lowRISC, and Google [413]. These shared efforts highlight the maturity of the technology and its potential for widespread adoption [414].

Despite the growing interest in CHERI, with a wide collection of materials over the past decade, including posters, conference proceedings, talks, repositories, and academic publications, a clear and systematized reference surrounding this technology is still lacking. While previous works have partially discussed capability systems and their influence on CHERI [164, 387], they do not provide a unified and complete reference for this technology. At present, it is hard to assess the different directions and areas of exploration of CHERI. Our work seeks to bridge this gap by presenting a comprehensive survey and systematization of knowledge on CHERI that incorporates multiple sources of information and examines CHERI from both hardware and software perspectives. This includes a detailed glance at hardware platforms, SoCs, and microarchitectural support, as well as the entire software stack, from low-level memory safety mechanisms

to higher-level applications. Furthermore, we trace CHERI evolution from its early concepts to its current role in enabling fine-grained memory protection and we outline potential future directions for research and development in the CHERI ecosystem.

This article is organized as follows. Section 5.2 provides a historical overview of capability-based systems, including hardware and software-based technologies. Section 5.3 outlines the concepts of the CHERI protection model. Section 5.4 describes the required hardware modifications and the current availability of CPUs, SoCs, and platforms. Section 5.5 explores the required modifications of existing software components and novel mechanisms introduced to leverage CHERI capabilities. Section 5.6 highlights the current adoption of CHERI in different domains, such as automotive, cloud computing, and IoT devices. Section 5.7 discusses future directions. Finally, Section 5.8 presents the concluding remarks.

5.2 Capability Systems: a Historical glance

In the mid-1960s, systems such as the Burroughs B5500 [144] and the Rice Computer [415] established foundational concepts for the formalization of capabilities. These architectures employed partitioned memory into distinct segments (*segmentation*). Each segment contained either data or code and was accessed via descriptors that encoded the segment's location together with the set of authorized operations. In 1966, Dennis and Van Horn [66] introduced a set of abstract instructions to provide memory protection for multiprogrammed systems, i.e., systems that execute multiple programs concurrently on a single or multiple CPUs. In their model, a program executes within a sphere of protection defined as a collection of capabilities. Each capability is an unforgeable authority token that confers access to a system object with specific permissions, such as read or write. The work of Dennis and Van Horn catalyzed the development of numerous capability-based architectures through the 1980s, as surveyed comprehensively by Levy [69]. For example, Lampson [70] and Fabry [142] formalized a hardware capability-based addressing scheme suitable for object-based programming, which in turn led to the emergence of a variety of other capability systems in subsequent years [72].

Despite their advantages, capability-based systems largely fell out of favor during the 1990s. Several factors contributed to this decline: (i) the complexity of capability management, particularly the operation of invalidating (*revoking*) capabilities to prevent their further use by programs, posed significant implementation challenges; (ii) the widespread adoption of RISC architectures [79, 80] and evolving market demands shifted priorities toward performance and efficiency rather than security; (iii) constraints on available silicon area limited the feasibility of dedicating hardware resources to capability-based designs; and (iv) pervasive misconceptions about the nature and advantages of capabilities relative to alternative security mechanisms [161], e.g., ACLs, which associate each system resource, for example a file, with an explicit list of permissions.

In the contemporary landscape, characterized by pervasive connectivity and large-scale distributed infrastructures, there is renewed interest in capability-based protection as a foundation for constructing

trustworthy systems capable of addressing modern security threats. In the following sections, we survey historical implementations of capability systems (purely hardware, purely software, and hardware–software co-designs) [416] that have informed the design of modern capability architectures, such as CHERI.

5.2.1 Hardware Capability Systems

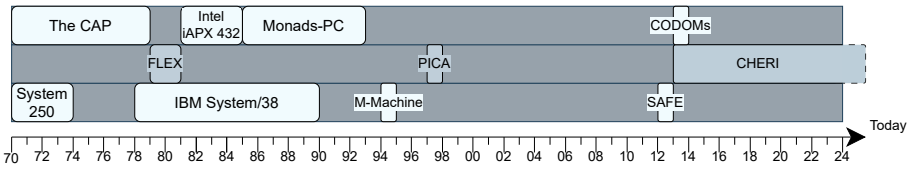


Figure 14: A timeline of significant hardware capability systems, ranging from the 1970 to today.

Figure 14 provides a timeline of the historical capability systems from their inception till today. Table 10 gathers the fundamental features of the most notable hardware capability systems. Given the scope of this survey, we highlight the CHERI architecture in the table and provide more details in Section 5.3.

5.2.1.1 Capability Storage

There are two principal architectural strategies for preserving the integrity of capabilities: (i) capability lists (*C-lists*) and (ii) tagged memory. A *C-list* is a form of capability storage that physically separates capabilities from ordinary data, e.g., by allocating distinct memory segments for data and for capabilities [142]. The Plessey System 250 [147, 148] and the CAP Computer [151, 152] are canonical instances of *C-list*-based architectures.

In contrast, tagged memory associates one or more metadata bits with each data word, thereby permitting the co-location of data and capabilities within a single segment. While *C-lists* can complicate object management due to the fragmentation of a single logical object across multiple segments, tagged memory simplifies software design and implementation by enabling the unified storage of object data and its corresponding capabilities within the same segment. Tagged capabilities were first investigated in systems such as Flex [153, 154] and IBM System/38 [74–76], and subsequently became widespread in capability-based architectures including the M-Machine [77], PICA [155, 156], and SAFE [157].

The Intel iAPX 432 [149, 150] attempted to combine the advantages of both paradigms by allowing data and capabilities to share a single segment that is logically partitioned into two regions. A per-object descriptor specified the sizes of the data and capability regions and thus determined the position of the boundary between them.

Finally, CODOMs [158] adopted an alternative mechanism that avoids both tagged memory and strict physical segregation. In this design, capabilities may only be transferred into pages that explicitly permit load or store operations. A hardware-maintained list, controlled by the TCB, enforces this policy, thereby regulating where capabilities can be fetched from or stored in memory.

System	Description	CS[1]	COR[2]	CR[3]	CPC[4]
Plessey System 250 [147, 148] (1972)	Commercial computer for real-time and fault-tolerant communication telephone-switch controller.	CL	S	no	Each protected procedure has a central table with all accessible data, code, and capability (*Central Capability Block (CCB)* segments. Procedure is accessible through an enter capability to the CCB.
The CAP Computer [152] (1977)	32-bit research capability system developed at the University of Cambridge from '70 to '77. It uses a dedicated 64-entry capability unit that automatically searches for and loads capabilities to access memory.	CL	S	limited	Requires an entry capability to a process-local table entry that changes the addressable data, code and capability segments.
Flex [153, 154] (1979)	System built by the UK Ministry of Defence's Royal Signals and Radar Establishment (RSRE) on the principles of capability-based security for highly secure and flexible computer systems.	TM	U	yes	N/A
IBM System/38 [74] (1980)	Mid-range computer designed for business computing applications. The System/38 implements capabilities as a combination of hardware and microcode.	TM	S	yes	One profile for each process with all available addressable objects and access permissions. Specifies mechanism to prevent profile sharing between the caller and callee.
Intel iAPX 432 [149, 150] (1981)	Commercial microprocessor designed to support advanced operating systems with capabilities for fine-grained access control and object-oriented programming. The iAPX 432 uses a combination of hardware and microcode to implement and abstract many operating system operations, like interprocess communication.	CL	S	yes	Supports domain objects as a collection of procedures that can access private objects. Consists of a list of capabilities to code and objects. A dedicated instruction is used to create the domain object that returns an entry capability to the protected domain.
Monads-PC [417, 418] (1985)	32-bit research capability-based machine which was built by David Abramson and John Rosenberg at the University of Monash. Monads-PC has the ability to support segments and paging.	TM	U	no	Uses a stack address space to store linkage information (bases), procedure-local data, and temporary data for expression evaluation. During a procedure call, the call mechanism adjusts the bases to establish the called procedure's execution environment.
M-Machine [77] (1995)	Proof-of-concept system that explores capability-based protection in a heterogeneous architecture to improve performance/area ratio. It implements a hybrid system for memory protection: (i) capability-based addressing system using Guarded Pointers for fine-grained memory protection; (ii) paging to support virtual memory and seamlessly relocate physical memory.	TM	U	limited	Uses entry capabilities that point to the domain code. Requires manual clearing of registers to prevent data and capability leakage.
PICA [155, 156] (1996)	Superscalar, 7-stage pipeline 32-bit MIPS processor. The PICA architecture adds capability-based access control to regular load-store RISC architecture with controlled hardware complexity and no impact on the processor's critical path.	TM	S	yes	The context for each protected domain is stored in a dedicated special stack that prevents capability and data leakage from one domain to another via stack or registers. Call/return sequence is ensured by an instruction tracker in hardware.
SAFE [157, 419] (2011)	Security-oriented processor that focuses on providing fine-grained privilege separation and object-based enforcement semantics at the architectural level.	TM	S	yes	Represents protected domains as a collection of three capabilities, i.e., code, data, and authority. To switch between domains, it uses atomic operations as simple as function calls, a.k.a. gates.
CODOMs [158] (2014)	CODOMs is a *code-centric* capability system that isolates multiple software components in a shared address space while minimizing the impact on existing software protection models.	APL	U	yes	Extends the page table with capability metadata and associates a tag to each memory page (*PTCaps*). An isolated domain is built by marking multiple pages (code and data) with the same tag. Each domain is isolated from another and communicates or transfers data via capabilities stored in capabilities registers (*CapRegs*) or spills capabilities in dedicated stack capabilities memory.
CHERI [37] (2014)	CHERI is an ISA-agnostic capability-base protection model that provides fine-grained memory protection (pointer-safety) and isolation primitives for application compartmentalization.	TM	U	yes	Defined protected domains via a sealing mechanism, entry points, and atomic domain transitions (see Section 5.3 for more details).

Table 10: Hardware capability-based systems. [1]Capability Storage - Tag-memory (TM), C-Lists (CL) or Access Protection Lists (APL), [2]Capability and Object Representation - Split (S) or Unified (U), [3]Capability Revocation - yes, no or limited, and [4]Capability-Protected Compartmentalization

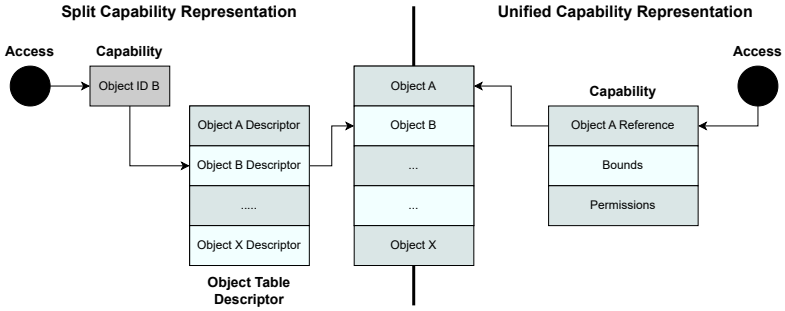


Figure 15: Comparison of unified and split capability representations schemes.

5.2.1.2 Capability and Object Representation

In Table 10, we categorize capability systems as *split* or *unified* according to their respective capability and object representations. Figure 15 provides a schematic overview of these two design paradigms. Split architectures (left side of Figure 15) were prevalent in early capability systems. In such architectures, object descriptors are maintained in centralized mapping tables, while capabilities contain unique identifiers that are used to reference these objects. Representative systems include the Plessey System 250 [147, 148], The CAP [151, 152], PICA [155, 156], and the Intel iAPX 432 [149, 150].

As an illustrative example, the CAP computer employs a multi-level indirection mechanism for capability resolution. Each process maintains a local object table that stores capabilities referencing indices in a global (central) object table. The index held in a process's local storage effectively serves as a pointer to a capability in the parent process's object table, with this chain of indirections continuing recursively until it reaches the root process.

To mitigate the performance overhead introduced by the implicit level of indirection in split capability designs, subsequent research trends focused on unifying capability and object representations by embedding object metadata directly within the capability itself, thereby obviating the need for a centralized object table (right side of Figure 15). Representative architectures such as the M-Machine [420], Monads-PC [417, 418], and SAFE [46] adopt a unified capability format that is amenable to storage in processor registers (e.g., the Guarded Pointer abstraction [420]). This representation typically encapsulates access permissions, an object identifier/reference, and spatial bounds describing the region of memory associated with the object. To further reduce storage and propagation costs, these systems frequently employ compression or encoding techniques that enable an efficient representation of capabilities both within the processor pipeline and in memory.

5.2.1.3 Capability Revocation

One approach to revoking capabilities involves using indirection by defining a *weak capability* that points to a *revoker capability* [421]. This *revoker capability* can later be used to invalidate all capabilities that reference it. Examples of this mechanism, include the CAP computer *relative capabilities* [151, 152]

and IBM System/38 *unauthorized capabilities* [74–76]. These systems simplify capability revocation, but exert significant pressure on hardware and caches during capability addressing. Instead, tagged architectures such as Plessey System 250 [147, 148] and M-Machine [420], lack efficient mechanisms for capability revocation and require a complete memory sweep through garbage collection to eliminate dangling capabilities (identified by the tag bit) to inactive objects.

Several systems realize capability revocation through coordinated mechanisms spanning both software and hardware. The Intel iAPX 432 [149, 150] supports capability revocation on the stack by disallowing the storage of capabilities in segments whose lifetimes exceed that of the stack frame, thereby ensuring that capabilities cannot outlive their intended scope. In the PICA architecture [155, 156], each object is assigned a unique identifier, which enables capability invalidation by deallocating the corresponding object and subsequently purging both the memory-to-capability mappings and the entries in the TLBs. SAFE [157] employs a dynamic tagging mechanism in which memory locations are tagged and runtime rules are enforced to regulate and invalidate capabilities as needed. In CODOMs [158], each capability is linked to a revocation counter stored in memory and embeds the counter value observed at the creation time. Incrementing the revocation counter invalidates all capabilities associated with it, providing efficient counter-based revocation.

5.2.1.4 Capability-protected Compartmentalization

All the architectures in Table 10 can compartmentalize programs into several fine-grained domains protected by capabilities, commonly including sealing and entry mechanisms. For instance, the M-Machine defines new architectural capability types: (i) sealed pointers, which are unforgeable pointers for building high-level software-defined policies and (ii) entry capabilities, which are opaque, unmodified pointers that can only be used to gate access to protected domains. PICA employs techniques to ensure no data is shared across boundaries, except for implicitly passed registers as arguments. SAFE defines protected domains as three capabilities: code, data, and authority and uses atomic operations, as simple as function calls, to switch between domains. CODOMS follows a hybrid design using a mix of capabilities and Access Protection Lists (APLs). APLs track which domains can access or invoke operations on other domains memory pages, and CODOMS checks these lists at runtime along with page-level permission to enforce cross-domain isolation efficiently.

5.2.1.5 Discussion

The Plessey System 250 [147, 148] is an example of a commercial computing system that effectively integrated capability-based protection mechanisms, extending its model to include I/O devices [69]. In contrast, the System/38 and Intel iAPX 432 [149, 150] faced commercial challenges: high costs and inadequate compiler optimization favored alternatives such as the 8086 and VAX-II/780 [422, 423]. The IBM System/38 [74–76] complexity in capability revocation led to its replacement by the IBM AS/400,

which eliminated capability support [78]. The Flex machine [153] offered robust security features, including compartmentalization and tagged memory, but failed commercially due to high costs and strong competition. The M-Machine [77], while a conceptual pioneer, failed to gain traction due to internal memory fragmentation and incompatibility with standard C idioms [79]. Despite this, its framework influenced modern capability systems like CHERI [37]. The SAFE [46] low-fat scheme for representing capabilities also has notable limitations, including the requirement for all pointers to remain in bounds, complicating pointer arithmetic and being incompatible with common C idioms [79, 81]. Finally, the PICA [155, 156] architecture's lack of adoption stems from insufficient virtual memory support and the absence of physical implementation, inhibiting its validation [79].

5.2.2 Software Capabilities

Table 11 summarizes the most notable capability-based OSs and microkernel architectures. These systems span several decades of research and development, from foundational work in the 1970s with Hydra [67] to modern implementations like seL4 [51] and Google Fuchsia [146]. The table introduces each system with a brief description of their contributions to security and performance in OS design. It also highlights if the project is still active. These systems may present differences on the organization of the capabilities and revocation mechanisms. Moreover, it is also important to consider the trade-offs between static and dynamic configurations that these systems offer.

5.2.2.1 Types of capabilities

Capabilities may be organized into explicit lists, hierarchical structures, or lightweight references. For instance, capabilities can be managed in a hierarchical derivation tree, as seen in seL4 [51] and Fiasco.OC [52]. In these systems, new capabilities are derived from existing ones with equal or reduced rights, enabling precise delegation and controlled propagation of access rights. Other systems, such as Hydra [67], present capabilities as *C-lists*, i.e., per-process tables indexing capabilities to system resources, representing an explicit and structured representation of a process authority. Composite [424] employs kernel-managed capabilities designed for modularity and inter-component delegation, i.e., capabilities are transferred and checked during component interactions. Alternatively, Google Fuchsia [425] uses *handles* as pointers to reference kernel-managed resources, and they rely on kernel policies to enforce access control. Persistent object stores, as leveraged by EROS [68], integrate capabilities with long-term storage, ensuring that access rights persist across reboots.

5.2.2.2 Revocation

Revocation ensures that access rights can be rescinded reliably and efficiently. A key distinction is whether revocation is *direct*, i.e., explicitly controlled via system calls, or *indirect*, i.e., automatically handled. Revocation strategies can vary along systems. Direct revocation in seL4 and Fiasco.OC employs a Capability

OS name	Author & Ref	Description	Types	Revocatio	Active
Hydra (1979)	Wulf et al. [67]	One of the first OSs focusing on capability-based security and resource management. It introduced capabilities to control access, influencing later secure OS designs like EROS and seL4.	- C-list	Indirect	☐
EROS (1999)	Shapiro et al. [68]	A capability-based microkernel OS featuring a single-level storage model and transparent persistence. It manages resources through capability invocations, optimizing memory, address translation, and process handling with virtual copy spaces and space banks, enabling efficient and secure access control.	- Hierarchical - Persistent object store	Direct	☐
seL4 (2006)	Heiser [51]	A formally verified, capability-based microkernel in the L4 family, ensuring system security and compliance across specific architectures and kernel versions. It features three capability types—i.e., kernel objects, abstract resources, and memory allocation—managed in per-thread *CSpaces*. With support for hardware virtualization and paravirtualization, seL4 can also function as type-1 hypervisor.	- Hierarchical - Explicit capability trees	Direct	☑
Composite (2008)	Parmer et al. [424]	An OS featuring hierarchical user-space schedulers, enabling flexible system partitioning through user-defined components. Unlike standard capability systems, it merges *Page Tables* and *Capability Tables* into a unified *Resource Table*, allowing dynamic capability management and operation execution.	- Kernel-managed - Modular	Indirect	☑
Fiasco.OC (2009)	L4Re.org [52]	Fiasco.OC, the microkernel for the L4Re project, is designed as a hypervisor meeting strict safety certifications like ASIL-B. It employs software-defined capabilities for controlled kernel object access and supports source code verification to ensure reliability. With `uvmm`, Fiasco integrates virtual machines, leveraging hardware virtualization and configurable isolation mechanisms for secure operation.	- Hierarchical - Explicit capability trees	Direct	☑
Google Fuchsia (2021)	Google Inc. [425]	Fuchsia uses the Zircon kernel—a secure, capability-based kernel for embedded systems. Zircon encapsulates system resources as objects (e.g., Channels, Processes, Threads) accessed via *Handles*, enforcing security through sandboxing and the principle of least privilege.	- Handles	Indirect	☑

Table 11: Software capability-based OSs.

Derivation Tree (CDT), which records parent-child relationships between capabilities. Revoking a capability from the tree invalidates all derived capabilities. EROS, as a persistent system, revokes capabilities by modifying or deleting the referenced persistent capability object. In contrast, more dynamic systems like Composite and Google Fuchsia prefer indirect revocation approaches such as *reference counting*—where

a capability is revoked when all references to the associated object are closed—, or *revocation lists* for Hydra.

5.2.2.3 Discussion: Static VS Dynamic capability configuration

The balance between static and dynamic configurations marks trade-offs in predictability, security, and adaptability. Static configurations define capabilities at system initialization, simplifying verification processes and eliminating runtime unpredictability. However, they lack flexibility to handle dynamic workloads. Conversely, dynamic configurations are flexible and easily adapt to different scenarios but, at the same time, increase the complexity of formal guarantees. Most of the systems presented use a hybrid approach, supporting both static configurations for predictability and dynamic configurations for runtime responsiveness or a combination of them. For instance, seL4 uses CapDL [426] to describe kernel objects and the distribution of capabilities among the objects; at runtime, capabilities may still be delegated and manipulated through kernel primitives, allowing the system to remain responsive to dynamic workloads while preserving strong confinement guarantees.

5.3 CHERI Protection Model

CHERI extends ISAs with capability-based primitives to improve security and prevent vulnerabilities. CHERI technology introduces two fundamental primitives: (i) fine-grained (byte-level) protection using capabilities that act as fat-pointers, encoding bounds and permissions to restrict memory accesses at the instruction level within a single, shared address space; and (ii) secure code encapsulation based on capabilities that enforce single-address-space software compartmentalization [163]. Table 12 outlines the core principles of CHERI, including the composition of the capability, core properties, hybridization and sealing mechanisms, and how potential capability misuses are handled.

5.3.1 Capability Composition, Core Properties, and Misuse Handling

In CHERI, capabilities are unforgeable (*integrity*) tokens of authority through which programs access all memory and services within an address space. Figure 16 depicts a possible CHERI capability for a 64-bit architecture. These fundamental ISA-defined hardware types can reside in memory or registers and replace traditional pointers. Memory is represented with robust primitives composed of an address and associated metadata, such as bounds and permissions. Additionally, a dedicated 1-bit validity tag (*validity bit*), stored in the system tagged memory, indicates whether a memory location holds a valid capability (if set to 1), otherwise, the value is interpreted as ordinary data. This ensures that ordinary data cannot be misinterpreted or misused as a capability (see Section 5.4.3). At system boot, only one powerful *primordial capability* with full permissions and memory exists. All other subsequent capabilities are derived from this primordial capability (*valid provenance*), with reduced or equal privileges (*monotonicity*). Consequently,

Principles	Description	Details
Capability Composition	Address + metadata (bounds, permissions, *otype*) + 1 out-of-band tag bit for integrity	Size: $2\times$ natural ISA address size + 1 tag bit
Core Properties	*Capability Integrity*: ensures that capabilities cannot be tampered without invalidation	Prevents attacks like capability forgery and unauthorized privilege elevation
	Bounds Checking: enforces only in-bounds operations	Prevents buffer overflows, stack corruption, and control-flow attacks
	Permissions Control: ensures that operations within a memory region are limited	"perm" field represents a bitmask where each bit provides a specific permission (such as load, store, and execute)
	Valid Provenance: requires all capabilities to be derived only from valid parent capabilities	Prevents the creation of invalid capabilities
	Monotonocity: guarantees that new capabilities inherit privileges $\leq$ to their parent	For example, transforming a read-only capability into a read/write capability is not allowed
Misuse Handling	Hardware Exceptions	Invalidates the capability and/or its tag bit
Hybridization	*Virtual Address*: complements the MMU role providing finer-grained memory protection	MMU for inter-process isolation $\cup$ CHERI in-address-space isolation
	Pointer: supports both legacy and CHERI-aware code	*hybrid-cap* and *pure-cap* modes
	Special registers: introduction of special registers to support hybridization	Such as Default Data Capability (DDC) and Program Counter Capability (PCC)
Sealing Mechanism	Enforces immutability and prevents unauthorized modification, dereferencing, or jumps	Object-based compartments VS *Sentry* capabilities

Table 12: Basic principles of the CHERI ISA, followed by a brief description and some additional details.

memory accesses are constrained by the bounds (*bound checking*) and permissions (*permissions control*) of the derived capability. *Misuse of capabilities* triggers hardware exceptions, invalidating them or nullifying their tag bit.

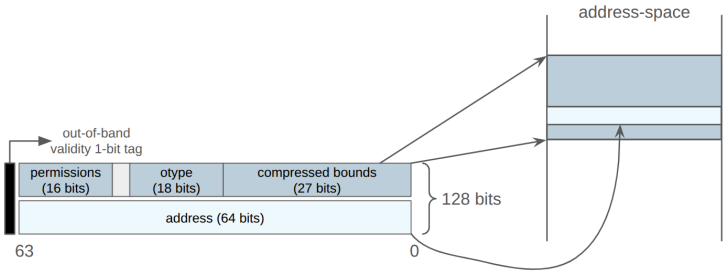


Figure 16: CHERI hardware capability 128-bit format representation in memory (simplified).

5.3.2 Hybridization

CHERI follows a hybrid protection model that complements traditional isolation techniques (like MMU-based isolation) rather than replacing them. The protection model supports both legacy and CHERI-aware code through special capability registers for memory and execution control. These registers include the

Default Data Capability (DDC), which defines the boundaries for memory access in non-capability code, and the PCC, which manages control flow for code execution. The introduction of capabilities and these special registers changes how pointers and function calls are represented, motivating a new application binary interface, the *CheriABI* [89], which defines the memory layout, calling conventions, and register usage. CheriABI-aware compilers, e.g., Clang/LLVM, allow code to be compiled in (i) *hybrid-cap* mode, where traditional pointers coexist with CHERI capabilities or in (ii) *pure-cap* mode, where all pointers are replaced with capabilities, including implied pointers (e.g., stack pointer and return address). In hybrid-cap mode, programmers annotate which pointers in the code the compiler should treat as a capability (through the keyword `__capability` [427]). This design decision enables incremental adoption of CHERI by progressively adapting and recompiling existing codebases.

5.3.3 Sealing Mechanism

The *otype* field in the CHERI ISA metadata enables the the *capability-sealing* mechanism [82, 428]. This makes a capability *immutable*, i.e., unauthorized modifications, dereferencing, or jumps invalidate the capability and raise a hardware exception. Without this mechanism, CHERI would not be able to support robust software compartmentalization in a single address space. The CHERI ISA supports two categories of sealed capabilities that allow non-monotonic operations: (i) Object Capability, sealed code and data capabilities that enable software compartmentalization within a single address space; and (ii) Sentry Capabilities, sealed code capabilities that serve as an entry point for atomic control-flow transition via jump instructions. Object capabilities involve a pair of sealed capabilities (one for code, one for data) with a software-defined *otype*, which can only be atomically unsealed and invoked via specific control-flow instructions (*CInvoke* instructions on RISC-V and *BLRS* on Arm Morello, see Section 5.4.1). Upon invocation, the unsealed code capability is loaded into the PCC register and the unsealed data capability is loaded into the DDC register. This allows compartments within a single-address-space to perform domain transitions without requiring supervisor (e.g., kernel) intervention. Sentry capabilities provide a secure way to jump to a function entry point in another compartment via standard jump instructions. Unlike object capabilities, sentries do not carry an associated data capability and are unsealed automatically during the jump operation. For secure inter-compartment data sharing, compartments can use sealed data capabilities: a capability sealed with a particular *otype* can only be unsealed (*CUnseal*) and dereferenced by a holder of the matching unsealing capability.

The ISA enhancements (such as newer instructions to manage the capabilities) provided by CHERI are the foundation of secure computing stacks. However, for applications to fully benefit from the CHERI approach, intermediate layers such as compilers, OSs, runtime systems, and libraries must be adapted to leverage hardware capabilities and provide enhanced security. Figure 17 depicts a pyramid chart where the position of each component reflects the extent of modifications required to support CHERI: the hardware, which requires the most significant modifications, is placed at the bottom, while applications, which require the fewest changes, are positioned at the top.

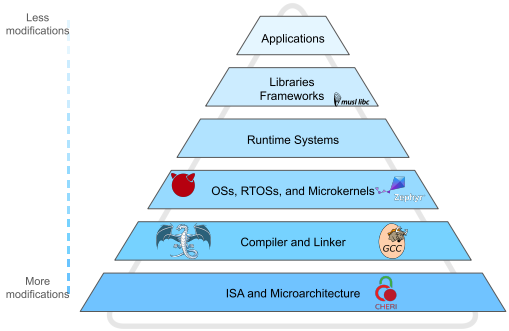


Figure 17: Pyramid chart illustrating CHERI impact across the computing stack, from the hardware layer requiring extensive architectural and microarchitectural modifications to the application layer, which requires minimal adjustments.

The following sections describe the state-of-the-art approaches for each layer, presented in a bottom-up fashion, starting from the CHERI hardware and progressing through the higher layers of the computing stack.

5.4 CHERI Hardware

CHERI is an ISA-agnostic protection model, meaning it can be mapped to multiple ISAs. Different ISAs require specific modifications to support the CHERI capability model, including introducing new registers, instructions, and tagged-memory support. Moreover, other components such as MMU, cache management, and exception handling require modifications. For instance, the MMU requires new page-table permission bits for loading or storing capabilities. Currently, the CHERI ISA supports all mainstream ISAs (e.g., RISC-V, Arm, x86, and MIPS).

5.4.1 CHERI-Aware ISAs

CHERI-MIPS was the first instruction set architecture to integrate the CHERI protection model in 2013. This enabled researchers to explore capability semantics and refine the CHERI protection model. For example, the synchronous domain call *CInvoke* [394] was implemented in CHERI-MIPS to replace the exception-based mechanism used for cross-domain calls (pair of instructions *CCall* and *CReturn*). *CInvoke* requires no trusted supervisor for cross-compartment calls and removes kernel complexity, while reducing runtime overhead. Such design decisions influenced the development of subsequent extensions to the RISC-V and Arm ISAs, specifically CHERI-RISC-V [394] and *Arm Morello* [167].

Over the last years, MIPS has seen a significant decrease in its presence in research and industry. In CHERI ISA version 9 [394], the University of Cambridge officially replaced CHERI-MIPS with CHERI-RISC-V as the primary reference architecture, reflecting the current industry trends. Several variants of CHERI-RISC-V have contributed to its evolution. For example, CHERIoT [87] is a fork of the CHERI-RISC-V ISA tailored for small embedded devices (32-bit systems). It simplifies the architecture at the cost of backward

compatibility by implementing a pure capability system. Notable features include (i) improved permission encoding, (ii) interrupt-guarded control flow, and (iii) a mechanism for temporal memory safety control. As of this writing, RISC-V is in the process of standardizing a collection of CHERI extensions [107] (*Zcheri extensions*), which incorporate feedback from various stakeholders, such as academia and industry. This effort includes merging the CHERI special capability registers with the privilege registers in RISC-V.

In the Arm architecture, the *Arm Morello* ISA [167] has laid the basis for the future adoption and growth of CHERI in mainstream commercial processors, potentially influencing broader industry standards. Several features added to the CHERI ISA by the *Arm Morello* ISA include a Compartment Identifier (CID) permissions bit and register, aimed at minimizing the side-channel attack surface in speculative execution resources (e.g., branch predictors). This ISA ties capabilities to specific compartments at the microarchitectural level, preventing information leakage between compartments. Lastly, while RISC architecture predominate in the CHERI landscape, a preliminary version of the x86-64 CHERI extension has emerged from CHERI ISA version 7 [429]. In this draft, the authors propose extending the general-purpose registers of x86 to serve as capabilities [430].

5.4.2 Capability Compression

Defining an efficient method to encode the capability metadata fields in memory and registers, such as the capability memory bounds, without hindering the deployment of CHERI technology has been a key focus in the CHERI literature [81, 431]. Early CHERI implementations, such as the CHERI-BERI CPU [432], encoded the capability metadata (bounds, address, permissions and type) into separate 64-bit words, making a capability $4\times$ the size of the architecture word size (see Figure 18). While CHERI-256 offers low-latency bounds checking and easier pointer arithmetic, it significantly impacts CPU datapaths and memory usage [81].

Inspired by the Low-fat encoding scheme [46], Woodruff et al. [81] proposed *CHERI Concentrate* as an alternative scheme for efficient capability encoding that represents capability bounds in a floating-point format relative to the current address. This requires only $2\times$ the architectural word size. The format shown in Figure 16 is denoted CHERI-128 for 64-bit systems. This encoding integrates efficiently with RISC single-cycle instruction set architectures and has become the de facto standard for capability compression in 64-bit CHERI processors [165, 166]. To mitigate the performance overhead associated with fully decoding bounds information under this scheme, CPUs typically employ a range of microarchitectural optimizations. CHERI implementations use multiple internal capability representation formats throughout the microarchitectural pipeline and progressively decode capabilities from their compressed in-memory format to a partial or fully decoded capability, as required at different stages of the pipeline. Performance-critical operations (such as pointer arithmetic) operate on only partially decoded state, primarily the cursor and the compressed bounds metadata, combined with single-cycle representability checks that act directly on the encoded capability representation. Full bounds decompression and verification is deferred to later pipeline stages or exception-handling paths, where precise bounds checks are performed off the critical

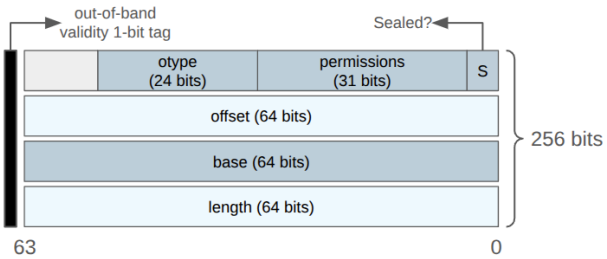


Figure 18: CHERI hardware capability 256-bit format representation.

path.

A later study in the CHERIoT [87] identified inefficiencies in the scaled-down CHERI-64 version for embedded systems due to precision loss and memory fragmentation issues, reaching up to 12.5% overhead. CHERIoT refined the CHERI-64 scheme by compressing permission bits, reducing object types, and improving precision, minimizing fragmentation to 0.19%. CHERIoT has been used to refine the *CHERI Concentrate* for RISC-V 32-bit CHERI standard specification.

Though most of the research on the CHERI concentrate compression scheme focuses primarily on RISC architectures, there is some work that formalizes the CHERI concentrate in ACL2 [433] for x86 CHERI architecture [430].

5.4.3 Capability Tagging

Efficiently handling and storing tags in memory requires special handling. In CHERI systems, there are two fundamental approaches to storing tags in memory: (i) repurpose the Error-Correcting Code (ECC) bits on DRAMs to store the tag bits, which is expensive and often not supported on low-power Random-Access Memory (RAM), and (ii) use a shadow bitmap table stored in a reserved area of memory on the DRAM, which requires a table lookup on every memory access and extra logic to update the tag table [431, 434]. Other solutions (not viable for production environments) are available, such as storing the tags in dedicated memory or distributing the cache tags per page [434]. The study conducted by Joannou et al. [434] implemented and characterized an efficient way to store tag bits for capabilities. Specifically, they proposed storing capability tags in an in-memory shadow table and tag cache near the memory. Tags are then fetched and updated in parallel to memory operations, ensuring that the validity of capabilities is checked and maintained simultaneously during memory accesses. For these scenarios, results show minimal overhead. Joannou et al. also defined a hierarchical scheme to compress tags based on the assumption that not every page will store capabilities.

5.4.4 CHERI CPUs, SoC, Platforms

Several actors provide reference implementations of CHERI CPUs targeting different classes of applications. In this respect, the University of Cambridge is particularly active. In Table 13, we summarize all

publicly available CHERI CPU implementations categorized into two types of processor classes: (i) small-scale MCUs for embedded applications with real-time requirements and no virtual memory support and (ii) APUs capable of running full-fledged OSs with virtual memory support.

5.4.4.1 CHERI Microcontrollers

There are several CPU implementations of CHERI technology in the MCU space. Piccolo [436] and Ibex [435] are representative examples of 32-bit RISC-V MCUs with CHERI extensions. Both cores support the execution of CHERI hybrid code to maintain legacy support. CHERIoT-Ibex [87, 437, 438] is another 32-bit CHERI-enabled processor that targets IoT devices. As part of the Portmeirion project [444] and based on the Ibex core, the Microsoft team characterized the challenges of porting CHERI technology to an MCU space, specifically for IoT applications. This resulted in a fork (*CHERIoT ISA* [445]) of the original Cambridge CHERI ISA. The CHERIoT-Ibex [87] is a pure capability core with no CHERI integer mode support, i.e., it can only run pure-capability code. Proteus [439] is another highly configurable 32-bit CHERI-RISC-V processor written in SpinalHDL.

5.4.4.2 CHERI Application Processors

Many application cores have been extended to support the CHERI technology. With the exception of the *Morello*-A76 processor [442], most implementations run as soft cores through FPGA emulation. BERI [432, 440] was the first implementation of CHERI technology in a MIPS-like ISA. The processor implements an in-order microarchitecture pipeline and uses a dedicated coprocessor for the CHERI ISA extension. CHERI-Flute [165] was the first RISC-V 64-bit application-class processor to support the CHERI-RISC-V ISA version 8 and to be formally verified [446] at the University of Oxford. Toooba [166] is another representative example of a high-performance RISC-V 64-bit out-of-order core with CHERI support. Moreover, Sá et al. [443] have extended CVA6, a highly configurable processor with hypervisor extension support [112], with the CHERI version 9 extension from Cambridge [394]. Table 13 shows a predominance of CHERI-RISC-V implementations with permissive licenses (APv2[1] and SP[2]) stemming from the open-standard nature of its ISA and the strong open-source community.

From an industrial perspective, Arm has also played a pivotal role in maturing the CHERI technology by launching the *Morello*-A76 processor. It features Armv8 ISA and CHERI ISA extensions and is deployed on a real platform, the *Arm Morello* board [83]. These contributions shed light on optimizing the ISA and the microarchitecture to balance security, performance, and hardware cost. The implementation targets the 64-bit design with MMU support.

Also on the industrial side, Codasip has been working on CHERI technology for RISC-V. In 2024, they launched the A730 [84], the first commercially available CHERI CPU for mid-end applications. It features a

[1] https://www.apache.org/licenses/LICENSE-2.0
[2] http://solderpad.org/licenses/

Name	ISA	μArch	MMU	Hybrid	Target	Evaluation		Lic.
						Performance	Area/LUTs	
CHERI Microcontrollers								
CHERI-Ibex [435] (2022)	RISC-V (RV32IMCB + M-mode + CHERIv8)	in-order 2- or 3-stage	☐	☑	FPGA	N/A	N/A	APv2
CHERI-Piccolo [436] (2018)	RISC-V (RV32IACDFM + M/U-mode + CHERIv8)	3-stage in-order	☐	☑	FPGA	CoreMark/MHz 4.1(10%) [164]	VCU-118 FPGA +62% [164]	BSD
CHERIoT-Ibex [437] (2023)	RISC-V (RV32IMCB + M/U-mode + CHERIoT ISA)	2 or 3-stage in-order + HW-assisted revocation	☐	☐	FPGA & ASIC	CoreMark/MHz 1.811(13.18%), worst case. perf 1.624 (21.18%) with load-filter for capability revocation [438]	TSMC 28nm HPC+ technology running at 300 MHz shows 2.15× the gates and 1.79× the power & max 5% in gates and power for HW-assisted revocation [438]	APv2
Proteus [439] (2023)	RISC-V (RV32IM + M-mode + CHERIv8 (no compressed capabilities)	5-stage in-order or super-scalar OoO	☐	☑	FPGA	N/A	Zynq Ultrascale+ XCZU9EG-2FFVB1156 FPGA +2.94 [439]	MIT
CHERI Application Class Processors								
BERI [432, 440] (2014)	64-bit MIPS + CHERI	5-stage in order + CHERI co-processor	☑	☑	FPGA	N/A	Altera Stratix IV EP4SGX230 FPGA +27% overhead (CHERI coprocessor + tag controller) [441]	APv2
Toooba [166] (2019)	RISC-V (RV64GC + CHERIv7)	superscalar OoO	☑	☑	FPGA	CoreMark/MHz 5 (-8%) [164]	VCU-118 FPGA +45% [164]	MIT & APv2
CHERI-Flute [165] (2019)	RISC-V (RV64GC + CHERIv8)	5-stage in-order	☑	☑	FPGA	CoreMark/MHz 3.6 (5.2%) [164]	VCU-118 FPGA +50% [164]	APv2
A76 (ARM Morello) [442] (2019)	Armv8-A + CHERIv8	OoO superscalar 13-stage	☑	☑	ASIC	SPECint 2006 benchmark running at 2.5 GHz: 5.70% (microarch. and ABI optimizations) and worst case perf. 28.01% (unmodified) [410]	<6% of the overall SoC [410]	C
A730 CHERI [84] (2023)	RISC-V (RV64G RISC-V Standard (ZCheri))	dual-issue, in-order, 7-stage	☑	☑	FPGA & ASIC	N/A	+4% [84]	C
CVA6 CHERI [443] (2024)	RISC-V (RV64GC + M/**HS**/VS/VU-modes + CHERIv9)	6-stage in-order	☑	☑	FPGA	N/A	+53.9%LUTs and +43.2%FFs compared to the baseline targeting Genesys2 FPGA running at 100MHz [443]	SP

Table 13: Public implementations of CHERI CPUs. For each CPU, we provide the following information: (i) name of the CPU and reference to the open-source implementation or other relevant sources; (ii) ISA; (iii) microarchitecture; (iv) virtual memory support (MMU); (v) CHERI hybrid execution modes support; (vi) when available: performance (CoreMark benchmark or SPECInt 2006), area (FPGA LUTs or gates), and power overheads; and (vii) license (Apache version 2 (APv2), MIT, BSD, Solderpad Hardware (SP) or Commercial (C)).

high-performance in-order pipeline and implements RISC-V standard CHERI ISA extensions (*Zcheri*) [107] with hybrid mode support.

5.4.4.3 CHERI-enabled Platforms

The *Arm Morello* board [83, 167, 447] is the first industrial prototype CHERI-enabled Arm SoC. It features four Arm application processors (i.e., Cortex-A76), multiple peripherals, and a complete memory subsystem with L1 and L2 caches, accompanied by a full software stack (e.g., toolchains) and a QEMU platform emulator. It aims to evaluate and gather feedback from stakeholders worldwide to promote the adoption of CHERI in real-world applications and to advance the maturity of the CHERI ecosystem.

The *Sonata* FPGA board [86, 448] is the first affordable open-source RISC-V CHERI capability-enhanced MCU platform designed for embedded systems and featuring a Microsoft CHERIoT processor. The platform incorporates essential onboard components, including timers, DMAs, Serial Peripheral Interfaces (SPIs), and Universal Asynchronous Receiver-Transmitters (UARTs). The memory subsystem consists of an L1 cache and a low-power on-chip HyperRam memory of 8 MiB. It also comes with a complete software stack from the CHERIoT project (e.g., LLVM toolchain,[3] debugger), as well as FPGA emulation setup and open-source board Printed Circuit Board (PCB) schematics [449]. The Sunburst chip [450, 451] is an open-source MCU chip design based on the Sonata system.

5.4.4.4 CHERI Overhead on Performance, Area and Power.

We acknowledge that measuring performance for prototypes involving CHERI technology can be complex, as the platforms may not be directly comparable due to differences in architecture and implementations. In addition, the currently available evaluations are limited in number and, in several instances, depend on synthetic benchmarks with restricted input datasets executed on FPGA-based prototypes. Such benchmarks tend to underestimate performance, as they do not comprehensively exercise critical microarchitectural components, including caches and TLBs. Consequently, these measurements should not be regarded as indicative of real-world performance results. Table 13 therefore summarizes the reported performance, area, and power overheads for CHERI-based processors, with the aim of establishing a baseline understanding of the implications of deploying CHERI technology in real-world systems.

An analysis on the reference CHERI RISC-V implementations [164], including Piccolo, Flute, and Toooba, reveals that the resource impacts, quantified in post-synthesis LUTs and Flip-Flops (FFs), ranges from 45% for CHERI-Piccolo to 62% for CHERI-Flute. The researchers suggest that as core size increases, the hardware impact decreases, as unaffected components like the floating-point unit dilute the modifications [164]. Similar trends were seen in other cores, e.g., on the CVA6H-CHERI [443]. In mature cores, such as the Morello A76, Codasip A730, and CHERIoT-Ibex [438], the impact is less than 6% of gates in the overall design [84, 87, 410].

From a performance perspective, CHERI introduces some overheads in core execution, particularly in pure-capability mode. CHERI requires additional instructions to manage capabilities (e.g., setting bounds and permissions) and to fulfill other related CheriABI requirements. As can be observed in Table 13, all cores perform between minimum 5.2% and a maximum of 28% worse than the baseline. CHERIoT

[3] https://github.com/CHERIoT-Platform/llvm-project

also exhibits similar results, showing a 13.18% performance overhead, primarily due to the narrower bus used for loading and storing capabilities, which necessitates two accesses. The worst-case scenario considers a hardware-assisted mechanism to perform capability revocation based on load-barriers, which increases the load-to-use latency of pointer accesses, resulting in more frequent pipeline stalls pushing the performance overhead to 21.18%. The only exception is CHERI-Toooba: Rugg et al. [164] explained that, in CHERI-Toooba, the CheriABI has fewer jump instructions, resulting in lower branch misprediction rates and improving performance. For *Arm Morello*, the reported performance overheads exhibit a strong dependence on both the selected ABI and the underlying microarchitectural configuration. The default configuration implemented in the FPGA prototype incurs substantially higher overheads (approximately 28%), primarily due to limitations in branch prediction, data-dependent exception-induced pipeline stalls, and increased pressure on the store queue. However, these overheads can be reduced to approximately 5.70% through ABI optimizations (e.g., mitigating branch stalls) and microarchitectural optimizations (e.g., enlarging store queues).

The surveyed CHERI CPU implementations across both embedded and application-class domains provide valuable insights into hardware integration efforts and trade-offs. In embedded systems, designs such as CHERIoT-Ibex [438], which adopt a pure-capability model, demonstrate that CHERI can enforce strong memory safety and compartmentalization with modest hardware overhead. Notably, CHERI can eliminate the need for traditional protection mechanisms such as the RISC-V PMP. For example, the CHERIoT RV32E + capabilities configuration incurs only a ∼3% increase in logic gates and ∼20% in power compared to an RV32E baseline with PMP, while offering significantly stronger security guarantees [438]. In more complex processors, such as the Morello [442] and Codasip A730 [84], the relative hardware cost of CHERI is further reduced due to amortization across large microarchitectural structures, including Floating-Point Units (FPUs) and superscalar pipelines. Reported performance overheads range from 5% to 28%, but these are also attributed to immature toolchains and the lack of CHERI-aware microarchitectural optimizations rather than just intrinsic ISA costs [410]. For instance, reports show that performance overheads in the Morello prototype (∼28%) can be reduced to ∼5.7% through targeted ABI refinements and pipeline-level enhancements [410]. These results yield the following implications for future CHERI implementations: (i) CHERI has the potential to fully supplant traditional protection mechanisms, such as RISC-V PMP, in microcontroller designs; (ii) the relative hardware overhead introduced by CHERI diminishes in more complex cores; and (iii) realizing the full potential of CHERI necessitates mature toolchains and pipeline-level optimizations, and therefore early performance overheads should not be interpreted as fixed costs of the ISA solely.

5.5 CHERI Software

While it is true that CHERI relies on specialized hardware, it is equally important to emphasize that modifications to the software architecture are required to fully take advantage of its benefits. Over the years,

several software efforts have been driven towards achieving complete CHERI-based systems. This section introduces and explores the diverse research fields of interest related to CHERI. We continue to follow the bottom-up approach illustrated in Figure 17.

5.5.1 Compilers and Linkers

Since the inception of CHERI technology, CHERI Clang/LLVM and LLD [82, 88] have become de facto compiler and linker for CHERI-aware platforms, including CHERI-MIPS, CHERI-RISC-V, and CHERI-Arm (*Arm Morello*) targets. Key modifications to the original Clang/LLVM implementation required to support CHERI include (i) the generation of suitable code to ensure capability properties, such as accurate bounds for stack allocations, (ii) the introduction of new C-language qualifiers for capabilities, (iii) the incorporation of capability-aware optimization, and (iv) the support of pure and hybrid capability modes (see Section 5.3 for details). CHERI Clang/LLVM also supports the Rust programming language (see Section 5.5.2.2). An interactive online resource is also available [452], allowing users to experiment with CHERI Clang/LLVM by exploring code snippets and viewing the compiled assembly.

CHERI Clang/LLVM served as the foundational toolchain for various research projects within the CHERI literature [453, 454]. For example, Tsampas et al. [453] developed a source-to-source compiler to automate compartmentalization in CHERI C code for CheriBSD, moving beyond manual code annotation. Similarly, El-Korashy et al. [455] created a preprocessing tool [456] to automatically preprocess and annotate CHERI C source code to define compartments. In addition to Clang/LLVM, other compilers have been adapted for CHERI, notably Arm's extended GCC [457] part of the GNU toolchain, which targets the Morello platform. This work is still in development. Additionally, Thibault et al. introduced SECOMP [454], an extension of the formally verified CompCert RISC-V [458] compiler for safety-critical applications, incorporating CHERI technology and experimental features for the CHERI ISA version 8.

5.5.2 Programming Languages and Tools

5.5.2.1 CHERI-C/C++

CHERI-C/C++ is a variant of the traditional C that uses CHERI capabilities to represent pointer types (e.g., pointers, return addresses, variables, and arrays). The adoption of CheriABI invalidates several assumptions commonly made by legacy software and libraries. In CHERI, programmers cannot freely cast between pointers and integers as in conventional C; such operations are restricted to prevent forging capabilities. Specifically, storing a capability as an integer (e.g., using `intptr_t`) results in the loss of its metadata and tag, making it unusable as a capability unless explicitly reconstructed and revalidated. Beyond pointer-integer conversions, CheriABI introduces additional constraints that affect legacy code migration. The increased pointer size (e.g., $128/64$ bits for capabilities versus $64/32$ bits for traditional pointers) can break data structure layouts, serialization formats (e.g., converting data structures to/from byte streams), and inter-process communication protocols that assume fixed pointers sizes. Capabilities also require

strict alignment (typically 16-byte alignment for a 64-bit architecture) which can cause issues with packed structures, custom memory layouts, and code that assumes pointers can be stored at arbitrary addresses (e.g., device drivers). CHERI-aware compilers can detect many of these issues at compile time, including invalid pointer-integer casts, alignment issues, and some instances of capability misuse. However, certain issues remain difficult to catch statically and may only manifest at runtime, such as unaligned capability stores in dynamically allocated or externally provided memory and provenance violations in complex pointer manipulation chains. Memory allocators and custom allocation schemes must be capability-aware to properly propagate bounds and permissions while respecting alignment requirements. Furthermore, optimization patterns, such as pointer arithmetic beyond object bounds, become invalid or require significant redesign. System calls and library interfaces that pass pointers through opaque integer types must be refactored to preserve capability provenance throughout the call chain. While these restrictions demand effort during the transition to CheriABI, compiler diagnostics significantly aid in identifying and resolving compatibility issues, with the remaining runtime concerns typically centered around dynamic memory operations and cross-boundary communication [89, 459–462]. We point readers to an (i) official document provided by the University Cambridge with the CHERI-C/C++ programming model [427] and (ii) a set of tutorials targeting QEMU and CheriBSD to get started with CHERI-C/C++ and CHERI in general [463]. Moreover, the Clang Static Analyzer (CSA) has been extended with checkers that can identify issues such as copying capabilities through under-aligned memory or a tag-stripping copy of capability [459, 464].

Cerberus [465–468] is an open-source tool that provides formal reasoning on the C memory model. Recent work has extended Cerberus to include CHERI-C semantics [469, 470], enabling the formalization of CHERI-C memory models to ensure program correctness and memory safety, while also formalizing the differences between CHERI-C and standard C and bridging the gap between CHERI-C capabilities and standard C usage. Another tool is Efficient SMT-based Context-Bounded Model Checker (**ESBMC**) [471, 472], which is an open-source bounded model checker tool for formally verifying C/C++, Kotlin, Python, and Solidity applications. Support for CHERI-C was recently integrated into the tool [473], but the implementation does not yet include a formally verified CHERI-C model and omits tagged memory support. Recently, Seung et al. [474, 475] formalized the CHERI-C memory model properties and semantics in the formal language Isabelle/Higher Order Logic (HOL) for use with verification tools such as ESBMC. They adapted the Gillian verification framework to prove the correctness of CHERI-C programs (*pure-cap* only) and to detect temporal memory issues such as dangling pointers.

5.5.2.2 CHERI Rust

The intersection of the Rust programming language and the CHERI architecture is being actively explored, focusing on compatibility, challenges, and potential optimizations [402, 476]. These works explore the integration of Rust compile-time restrictions with CHERI runtime enforcement, enabling *unsafe* Rust code to run safer. Studies have also evaluated CHERI ability to prevent bugs like out-of-bounds writes in the standard Rust library [476]. Other works have focused on CHERI Rust evaluations on the *Arm Morello*

platform. The results show differences based on `usize` interpretations: (i) a 64-bit address integer or (ii) a 128-bit capability integer, highlighting the performance impact and the requirement of additional operations for LLVM intrinsics (like *memcpy* and *inttoptr*) of the latter [402].

5.5.3 OSs, RTOSs and Microkernels

Name	Architectures			CAP-mode		MMU	RT[1]	EP[2]	References
	RISC-V	MIPS	Arm	Hybrid	Pure				
General-Purpose Operating Systems									
CheriBSD	☑	☑	☑	☑	☑	☑	☐	QEMU, FPGA, *Arm Morello*	Davis et al. [89]
CHERI Linux	☑	☐	☑	☑	☑	☑	☐	QEMU, *Arm Morello*	Wang et al. [477]
MMU-less CHERI Linux	☑	☐	☐	☐	☑	☐	☐	QEMU	Almatary et al. [478]
Real-Time Operating Systems									
CheriRTOS	☑	☑	☐	☐	☑	☐	☑	QEMU, FPGA	Xia et al. [388]
CHERI-Zephyr	☑	☐	☐	☐	☑	☑	☑	QEMU, FPGA	Jackson [103]
CHERIoT-RTOS	☑	☐	☐	☐	☑	☐	☑	Sail, FPGA	Amar et al. [479]
Microkernels									
FlexCAP	☐	☐	☑	☑	☑	☑	☑	*Arm Morello*	Olivier et al. [480]
CheriOS	☑	☑	☐	☐	☑	☑	☑	QEMU, FPGA	Esswood [387]
seL4	☑	☐	☑	☑	☑	☑	☑	QEMU, *Arm Morello*	Heiser et al. [481]

Table 14: List of CHERI-aware OSs, RTOSs, and Microkernels. With: [1]Real-Time and [2]Evaluation Platforms

In the context of CHERI, components such as OSs, RTOSs, and microkernels must be adapted or redesigned to benefit from the CHERI memory model, minimizing the performance degradation. Porting these components to the CHERI ecosystem involves addressing the CheriABI compatibility challenges discussed in Section 5.5.2.1, which are particularly marked at the kernel-level where code frequently interacts with hardware, manages low-level memory layouts, and implements core system services. Moreover, OS porting must ensure backward compatibility with legacy applications, optimize performance and security trade-offs for specific use cases, and adapt user-space components alongside kernel modifications. These requirements often necessitate new design strategies and frameworks (see Section 5.5.5).

This section reviews the CHERI community's efforts on implementing or porting these stacks into CHERI. We focus on key criteria for comparison such as the types of systems (i.e., General-Purpose Operating Systems (GPOSs), RTOSs, and microkernels), the architectures supported (i.e., MIPS, RISC-V, Arm), and the capability modes adopted. As further criteria, we also focus on the use of MMU-based isolation in addition to CHERI. For each work, we identify whether the technology is suitable for real-time targets and report the evaluation platform. Table 14 summarizes the analysis.

5.5.3.1 General-Purpose Operating Systems

Adapting GPOSs to the CHERI architecture is essential for demonstrating its security and performance benefits in practical scenarios. Moreover, GPOSs provide a familiar environment for a broader community, making it easier to port applications to the CHERI ecosystem.

CheriBSD [89, 99] is a modified version of FreeBSD with awareness of CHERI capability. As of this writing, CheriBSD is the most mature OS that supports CHERI, making it the primary choice for porting applications to the CHERI ecosystem. CheriBSD supports several architectures, including *Arm Morello*, CHERI-RISC-V, and CHERI-MIPS, and runs on multiple platforms and emulators. The CheriBSD kernel supports hybrid and pure-cap compilation. From a library perspective, several libraries, including specific ones for compartmentalization support, have been ported to CHERI (see Section 5.5.5). From an ABI perspective, CheriBSD supports legacy code and a CHERI-aware ABI. CheriABI [89] is the implementation of pure-capability processes, kernels, and applications that completely replace pointers with capabilities to manage memory, process creation, dynamic linking, and system calls.

CHERI Linux: Linux version 5.18 was ported to CHERI RISC-V through joint efforts by Huawei and the Fraunhofer AISEC [100, 101, 477]. The port includes minimal modifications to the Linux runtime system, drivers, and user-space applications. The kernel is suitable for hybrid or pure-cap compilation. The authors reported that most of the issues were resolved by addressing the problems raised by the compiler during compilation. Moreover, they used the Juliet Test Suite [482] to evaluate the security improvements (the adapted version includes 546 test cases): CHERI Linux mitigates all spatial memory exploits (152 test cases). The authors included a performance evaluation with Dhrystone, Core-Mark, and MiBench (average overhead: 16.3%, 14.4% and 3.7%, respectively). A work-in-progress on *CHERIfying* Linux is also being undertaken by Arm for the *Arm Morello* and by the *CHERI Alliance* and *Codasip* for CHERI RISC-V.

Almatary et al. [478] use CHERI-C and compartmentalization to enhance security of an **MMU-less** version of CHERI Linux by partitioning the kernel into isolated compartments—i.e., kernel modules, user applications, and libraries—reducing the impact of potential vulnerabilities. As the research is still a work in progress, validation and development are being conducted using QEMU. An evaluation on real platforms is reserved for future work.

5.5.3.2 Real-Time Operating Systems

Table 14 also reports CHERI applicability to real-time environments. The evaluations focus on properties of real-time systems such as resource utilization, reliability, and predictability.

CheriRTOS [388, 483] was specifically designed to provide a proof-of-concept for embedded systems, executing on a single-address-space physical address model (*MMU-free* system). CheriRTOS is an extension of FreeRTOS that supports CHERI-MIPS and CHERI-RISC-V. The FreeRTOS kernel was modified to manage CHERI capabilities in pure-cap mode, such as PCC, DDC, stack, and context switch. This allows for compartmentalized multitasking and fine-grained memory protection, while maintaining real-time guarantees. The authors compare the performance of the proposed CHERI system with a baseline system running without CHERI. The evaluation shows that the security benefit of the CHERI system outweighs the low overhead introduced (4.7% to almost no overhead).

CHERI-Zephyr is a port of the Zephyr RTOS to the CHERI-RISC-V architecture, developed through recent efforts by the University of Birmingham and the CAP-TEE project [103, 484]. Zephyr supports MMU and is written in C language, exposing it to common C memory-safety bugs [485, 486]. CHERI-Zephyr demonstrates how CHERI capabilities can mitigate these vulnerabilities, including safely handling known vulnerabilities [487, 488] via CHERI exceptions. The co-existence of CHERI and MMU is not addressed in this work. To the best of our knowledge, the development is still in the early stages and no performance evaluations are available.

CHERIoT-RTOS [479, 489], rather than retrofitting protection into an existing RTOS, decomposes core OS functionality into explicitly trusted and untrusted components from the ground-up. Running on CHERIoT-Ibex core, the system uses a dedicated loader to establish initial capabilities for each compartment, while a switcher handles context switches and cross-compartment calls. Notably, even the scheduler runs in a separate, isolated compartment rather than in the kernel itself. As with other RTOSs discussed, the key challenges are maintaining real-time predictability and low overhead while enforcing fine-grained protection; CHERIoT addresses them by keeping the TCB small and moving policy and high-level abstractions into unprivileged libraries.

5.5.3.3 Unikernel and Microkernels

Olivier et al. [480] argue that the unikernel OS model needs to incorporate intra-unikernel isolation to enhance security. Without such isolation, unikernels are vulnerable to internal threats. By leveraging CHERI, unikernels can enforce strict boundaries between subsystems within the same instance, reducing the impact of potential vulnerabilities without compromising their efficiency.

FlexCap builds on FlexOS and leverages *Arm Morello* hardware features to implement advanced isolation mechanisms [490, 491]. FlexOS [492] is a library OS that emphasizes customizable isolation and security measures. FlexOS integrates with Unikraft [493], a high-performance unikernel for isolated environments (e.g., using the MMU) and single-application operating systems. FlexCap [494] uses a

CHERI-aware version of Unikraft to handle capabilities for key primitives such as context switches, exceptions, and device access. FlexCap combines FlexOS and Unikraft to build secure and efficient systems in a compartmentalized environment using CHERI capabilities.

CheriOS [387] is a capability-based microkernel that uses CHERI to minimize the Trusted Computing Base and increase memory protection. It was implemented on CHERI-MIPS and CHERI-RISC-V in pure-cap mode. CheriOS follows a mutual distrust model, i.e., the OS and user applications do not trust each other. Only two components are considered trusted: the CHERI ISA and the *nanokernel*. The *nanokernel* provides essential primitives and enables the creation of compartments with distinct authority, allowing for fine-grained control over privileges. Additionally, the *nanokernel* uses both MMU and capabilities to manage and secure the state of every physical and virtual page.

seL4 [51], a formally verified microkernel, has explored CHERI integration to enhance userspace safety, as shown at the seL4 Summit 2023 [102, 481]. A recent work [495] introduced an seL4 version for CHERI-RISC-V, supporting pure-cap userspace and a hybrid-cap kernel, passing the *sel4test*.

5.5.4 CHERI Runtime System

The runtime system provides the environment to run an application as specified by the programmer. Runtime systems can implement different functionalities such as memory management (e.g., stack and heap management), threads and processes, OS interactions, exception and error handling, code execution, I/O operations (e.g., network and disk), and dynamic library loading. This section surveys the state-of-the-art literature of enabling CHERI technology in the runtime system, with focus on: (i) efforts to port programming languages runtime systems (e.g., memory allocators) to CHERI and (ii) capability revocation techniques for heap and stack temporal memory safety.

5.5.4.1 CHERI-aware Memory Allocators

Memory allocators are responsible for dynamic memory allocation and deallocation during program execution. Porting memory allocators to CHERI involves careful management of capabilities. Bramley et al. [496] assessed the status of CHERI-compatible memory allocators, uncovering vulnerabilities in both CHERI-aware and non-CHERI-aware compilers. For example, using *jemalloc* [497] can expose hidden memory when resizing capabilities bounds without zeroing out previous contents. There are three notable CHERI-aware forks of well-known memory allocators: *jemalloc* [497], *snmalloc* [498], and *dlmalloc* [499]. *jemalloc* is the default memory allocator for CheriBSD OS. Microsoft employs a hardened version of *snmalloc* in the CHERIoT [87] project.

5.5.4.2 Capability Management

The runtime system plays a crucial role in creating, propagating, and revoking capabilities during application execution. Cross-domain propagation of capabilities requires careful handling to prevent leaks and

Technique	Description	Key features
	Heap Memory Safety	
CHERIvoke [391] (2019)	Extends the memory allocator to mark deallocated memory regions in a bitmap shadow table and keeps these regions in a quarantine buffer. Once this buffer is full, a full sweep locates and invalidates capability references to deallocated memory regions recorded in the shadow table.	Implements a special load tag instruction to load multiple tags at once (*CLoadTags*) and adds page-level capabilities tracking bits to indicate if a page holds one or more valid capabilities (capability dirty bits).
Cornucopia [392] (2020)	A sweeping capability revocation system built atop CHERI and the CHERIvoke algorithm [391] implemented on CheriBSD.	Implements the hardware extensions proposed on CHERIvoke [391] in a 64-bit CHERI MIPS-base soft core.
Capability Boehm [500] (2022)	The Boehm-Demers-Weiser Garbage Collector (BDWGC) is a conservative GC widely used in C and C++ programs for automatic heap memory management. This adaptation allows BDWGC-based applications to run on CHERI systems.	It enables automatic heap memory management and safe reclamation of unused objects, enforcing memory safety via capabilities.
Cornucopia Reloaded [393] (2024)	Re-implementation of the Cornucopia [392] on *Arm Morello* with a complete redesign of core algorithm [391]. Reloaded embodies new architectural features to limit the effect of long global stalls and reduce the DRAM traffic, a.k.a stop-the-world-pauses, on each revocation pass found in previous techniques [391, 392].	Repurposes hardware extensions proposed on CHERIvoke [391] and adds per-PTE bit flag to trap all capability loads (*load barriers*).
	Stack Memory Safety	
Local capabilities [82] (2015)	Control flow technique that limits capability storage in constrained memory regions (typically the stack) and registers.	Adds two flow-control bits stored in the capability metadata (*global capabilities bit* and *local capabilities bit*).
StkTokens [501] (2019)	Formalizes a secure calling convention for cross-domains in CHERI based on linear capabilities to enforce local state encapsulation (LSE) and well-bracketed control flow (WBCF) in a single shared stack between two untrusted domains. Enforces revocation upon domain crossing.	Utilizes local capabilities [82] concept.
Tsampas et al. [453] (2019)	Ensures temporal stack memory safety for stack references in an abstract capability-based machine. It adds multiple locality levels to prevent capabilities from being stored in a memory with a longer lifetime cycle than their own.	Extends the capability metadata with n locality permission bits and adds dynamic checks to enforce locality when storing capabilities in memory.
Georges et al. [502] (2021)	New formally proved calling convention written in Coq that expands the idea of local capabilities and uninitialized capabilities to share and revoke capabilities.	Adds a new type of architectural capability (unitialized capability) that divides its authority into two memory regions: (i) read-write regions and (ii) write-only regions. System model in CHERI-MIPS-based simulator.
Georges et al. [503] (2022)	Secure calling convention that ensures LSE, WBCF, and stack temporal and spatial memory safety. It proposes using direct capabilities along with uninitialized capabilities [502] to enforce stack memory safety without requiring expensive full stack clears upon return/calls between two domain boundaries.	Adds new architectural capability type, *direct capabilities*. These capabilities can only be stored in the memory above, where it can read memory.
Borrowed capabilities [504] (2022)	A novel novel capability-revocation technique based on the concepts of linear capabilities and the idea of borrowing mutable and immutable objects in Rust language	New capability representation (borrowed capabilities) model in Sail [446] and CHERI-RISC-V.
	Heap and Stack Memory Safety	
CHERIoT [438] (2023)	Develops hardware-specific mechanisms for complete temporal heap and stack memory safety tailored for the embedded system space.	Revocation uses 1-bit per capability-size granule, enforced by a load barrier, and is assisted by a MMIO device. While *stack watermarking* enforces stack memory safety.
Conditional Capabilities [505] (2024)	Proposes a CHERI ISA extension to express fine-grained memory policies that prevent undefined behavior or uninitialized memory (e.g., *write-before-read*).	Adds permission bits to the capability metadata to express conditions. Moreover, it adds new instructions to set these conditions and operation bounds.

Table 15: CHERI capability revocation techniques

protect data integrity and confidentiality. The complexity and overhead of capability revocation can limit the adoption of CHERI, prompting research into secure, efficient revocation and propagation techniques. We group the contributions into three categories: (i) heap memory safety, (ii) stack memory safety, and (iii) works that address both of them. Table 15 provides a brief description, key features, and references of each contribution.

Heap memory safety ensures the correct management of dynamically allocated memory, avoiding, for instance, memory leaks and buffer overflows [391–393, 438]. Notably, Cornucopia Reloaded [393] uses a load barrier mechanism to dynamically track loaded capabilities, avoiding costly global operations and improving performance. The CHERI port of the Boehm-Demers-Weiser Garbage Collector [500] is exploring automatic heap management on CheriBSD for CHERI RISC-V and *Arm Morello*, tackling challenges such as pointer-shape issues and provenance. In the embedded space, CHERIoT [438] defines a set of mechanisms based on the addition of revocation bits to each heap memory granule, load-barrier filters or a background hardware worker to invalidate the capabilities pointing to freed memory.

Stack memory safety focuses on preventing issues such as stack overflows, unintended overwrite of local variables, and improper usage of function call frames. Several techniques exist to limit capability propagation in CHERI-based systems and to efficiently revoke ephemeral capabilities shared across domain boundaries [82, 438, 453, 500–503, 506]. To ensure stack temporal safety when invoking an untrusted domain, the system must uphold two properties: (i) Well-Bracketed Control Flow (WBCF), which ensures that the callee returns control to the caller and (ii) Local State Encapsulation (LSE), which guarantees that the caller's local state cannot be altered by the callee. The most innovative works include Tsampas et al. [453], which proposed a unique locality-based approach, enhancing CHERI capabilities with locality permission bits and dynamic checks to prevent capabilities from outliving their intended scope. Georges et al. [502] and Georges et al. [503] propose two methods to improve stack temporal memory safety. The first method [502] expands the notion of *local* and *uninitialized capabilities* to facilitate capability sharing and revocation using stack *watermarking* (similar to the method proposed by CHERIoT [87]). The second method [503] establishes a secure calling convention that combines uninitialized and direct capabilities. Skorstengaard et al. [506] advanced the concept of *linear capabilities*, converting stacks and sealed return capabilities into single-instance entities, ensuring safe delegation and centralized stack sharing. Vercammen et al. [504] proposed *borrowed capabilities*, a flexible mechanism that ties capabilities to lifetimes, allowing temporary breaches of linearity while ensuring safe revocation through lifetime tokens. Recently, Merve et al. [505] propose the idea of using *conditional capabilities* to express memory policies that could improve heap and stack memory safety. For instance, the *conditional capabilities* can be used for a *write-before-read* policy, requiring at least one write operation to the memory location pointed by the capability before granting read access.

5.5.4.3 Case Studies of CHERI Runtime Systems

Numerous studies demonstrated the feasibility and security benefits of porting real-world runtime systems, applications, and operating systems to CHERI [82, 89–94].

Watson et al. [82] introduced the first CHERI-C userspace runtime, allowing CHERI-C applications to operate in hybrid mode (i.e., supports non-CHERI libraries). CheriABI [89] later extended the runtime system for pure-capability applications, maintaining a hybrid layer for backward compatibility.

Several other runtime systems have also leveraged CHERI technology. For instance, Lowther et al. [90, 507] hardened MicroPython [508] for improved runtime security on the *Arm Morello* and Sonata platform. Gutstein [91] adapted the JavaScriptCore (JSC) interpreter for *pure-cap* code support. CHERI JNI [92] utilized the CHERI capability model to optimize buffer sharing (offer direct access) in the Java Virtual Machine (JVM) and prevent type-confusion attacks through sealing mechanisms. Additionally, Butcher et al. [93] explored integrating CHERI primitives with Ada [509] runtime checks to safeguard uncovered parts of the code or unsafe code in *pure-cap* code. The evaluation was performed for the *Arm Morello* platform in *pure-cap* mode. Patel et al. [94] optimized kernel page fault handling using CHERI capabilities to enhance pointer identification and accelerate page prefetching in pointer-intensive applications.

5.5.5 Libraries and Frameworks

A suite of libraries and frameworks has been developed to fully make use of CHERI capability-based security model. These components bridge lower-level system infrastructure and application-level software, ensuring that the benefits of CHERI principles are implemented across the entire software stack.

musl-libc [105] and **newlib** [104] are lightweight C standard libraries that have been adapted to work within CHERI ecosystems. These libraries provide essential runtime support for creating containerized or baremetal applications. Another set of libraries includes **libreSSL** [510] and **libssh2** [511], which are cryptographic and communication libraries.

CompartOS [512] is a framework and linkage-based compartmentalization model designed to leverage CHERI capabilities into an MMU-free, single-address-space and single-ring-privilege environment for isolating different linkage modules. CompartOS uses an adapted FreeRTOS (CheriFreeRTOS) kernel, extends the *libdl* library (provides functions for runtime loading and symbol resolution of shared libraries) for CHERI-aware dynamic linking, and leverages the LLVM compiler to limit the visibility of global variables. CompartOS has been compared against three different architecture features, i.e., without protection, PMP, and PMP+MMU. The extensive evaluation [512] analyzes the trade-offs of hardware overheads and security benefits, and highlights that the flexibility of CompartOS only incurs a slightly increased overhead compared to a full pure-cap solution.

CAP-VMs [513] is a VM-like abstraction that uses CHERI to isolate applications through memory capabilities. Its architecture consists of three main components: the Host OS kernel, the Intravisor, and the Capability Virtual Machine (cVM). CheriBSD serves as the Host OS kernel, providing essential primitives for synchronization, execution contexts, and I/O operations. The Intravisor runs as a process

within CheriBSD and handles the isolation and lifecycle management of cVMs. It distributes memory capabilities to cVMs, facilitates communication, and has access to the memory of all cVMs. The cVM, on the other hand, represents an isolated application, e.g., a program binary and its libraries, that runs as an isolated thread of the Intravisor process. At the time of this writing, two extensions of CAP-VMs exist: (i) Object Reuse with Capabilities (ORC) [514] extension, which improves the memory utilization of the overall framework and (ii) Sandbox fork caching mechanism [515], that reduces cold start latency of the cVMs.

5.5.6 Applications

The CHERI ecosystem has demonstrated significant progress in porting and adapting a wide range of real-world software applications to leverage enhanced security and performance benefits. We refer interested readers to the CheriBSD packages for the exhaustive list of applications ported or supported [516].

Furthermore, multiple tools have been developed for emulation, simulation, validation, debugging, and visualization. These tools address the challenges of limited hardware availability and provide robust environments for development and testing. As summarized in Table 16, emulator tools like QEMU [95] and Arm Fixed Virtual Platform (FVP) [517] simulate the behavior of a complete system, allowing the execution of multiple OSs (see Section 5.5.3). Tools such as Spike [96], Sail [233, 446, 518–520], VeriCHERI [521], and TestRIG [390] focus on the verification process of the ISA. CheriPlot [97, 522] provides graphs of the derivations of capabilities. Finally, the GNU Debugger (GDB) [523] has been extended to support CHERI, thereby facilitating debugging of CHERI-adapted software.

Name	Category	Architectures			Description
		RISC-V	MIPS	Arm	
QEMU [95]	System Emulator	☑	☑	☑	A generic and open-source system emulator that supports full system emulation, including some peripherals.
Arm FVP Morello [517]	Virtual Platform Emulator	☐	☐	☑	A software model of *Arm Morello* hardware for prototyping, development, and testing. It allows full system emulation.
Spike [96]	ISA emulator	☑	☐	☐	The official RISC-V ISA simulator, focused on emulating only the RISC-V instruction set.
Sail [233, 446]	ISA Formal Verification Emulator	☑	☑	☑	A formal specification tool for ISA semantics with an executable model. Sail models have been used for ISA development, implementation, and formal verification via automatically generated tests.
VeriCHERI [521]	Formal security verification tool	☑	☐	☐	A tool for formal security verification at the register transfer level (RTL). As a consequence, this tool is ISA-independent and uses SystemVerilog Assertions (SVA) properties to verify the security of the architecture.
TestRIG [390]	Testing Tool	☑	☐	☐	This framework tests RISC-V processors using random instruction generation. Using this tool, together with Sail models, allows automated verification tests.
CheriPlot [97, 522]	Visualizatior Tool	☐	☑	☐	A graphical tool for visualizing and tracking the capability flow at the user and kernel level.
GNU debugger (GDB) [523]	Debugger	☑	☑	☑	GDB allows users to debug CHERI-enabled software. It can inspect capability registers and display their content in a human-readable format.

Table 16: CHERI emulators, tools, and debugger

The experiences of porting software to CHERI reveal insights that can guide future adoption efforts. CheriABI compatibility issues follow consistent patterns: pointer-integer conversions, assumptions about pointer sizes, capability alignment constraints, and provenance violations. CHERI-aware compilers catch most problems at compile time, with runtime issues primarily in dynamic memory and cross-boundary communication. In practice, porting effort is usually modest; the challenge is locating issues rather than fixing them. In the CheriBSD package ecosystem [516], many applications require only recompilation once core libraries (e.g., libc and libc++) are adapted. Kernel-level components need more targeted changes due to direct hardware interaction and custom memory layouts, but modifications remain localized, as seen in the FreeBSD kernel port. Compilers and interpreters are notable exceptions. Compilers must be adapted to generate CHERI-aware code, ensuring that instruction selection, calling conventions, and memory operations comply with CheriABI, while interpreters face challenges from dynamic memory, code generation, and provenance tracking. This complexity explains why SPECint 2006 benchmarks requiring the compilation of the Perl interpreter (400.perlbench) and GCC compiler (403.gcc) were excluded [410]. For new CHERI adopters, we recommend checking dependencies against CheriBSD ports, monitoring pointer arithmetic and custom allocators, leveraging compiler diagnostics, and, if many dependencies need to be ported, starting with hybrid capability mode.

5.6 CHERI Application Domains & Use-cases

This section explores key application domains where the potential of CHERI is being actively investigated. We focus on two very relevant areas: edge-to-cloud systems and MCSs.

5.6.1 CHERI for Edge-to-Cloud systems

Edge devices form the backbone of distributed systems and cloud-based infrastructures, yet they face significant challenges in security and resource efficiency.

The IoT sector, in particular, represents a promising direction for CHERI adoption. In fact, these systems are small, cheap, and usually lack security hardware or software solutions, e.g., an MMU and virtualization. Additionally, the turnover of IoT applications is fast, creating opportunities for incorporating CHERI-based software. Moreover, many IoT applications fall within the domain of Cyber-Physical Systems (CPSs), which consist of distributed networks of interconnected components such as sensors, actuators, controllers, and computing devices. CPS systems face multiple challenges related to coordination, synchronization, communication, security, and reliability. For industrial IoT sectors, where security and long product lifetimes are critical, CHERI has emerged as a promising architecture to achieve robust security while keeping almost unmodified source code.

Safronov et al. [524] conducted an analysis of various CVEs related to wireless gateways, which are key components in large-scale IoT systems, facilitating device interconnectivity. Their findings indicate

that implementing memory protection techniques can eliminate 74% of these CVEs, increasing the overall security. Dodson et al. [525] examine two specific scenarios, a robot and a common Industrial Control System (ICS), where they integrate CHERI capabilities and Macaroon tokens (`libmacaroons`) [526], leveraging their structural similarity. Dodson [527] expands on the exploration of CHERI for CPS, introducing another use-case by integrating `libmodbus` into CheriFreeRTOS from the CompartOS work.

The transition from on-premises infrastructure to edge-cloud computing offers a number of advantages, including reduced costs, enhanced operational scalability, and increased reliability. Nevertheless, concerns regarding the privacy of data and the necessity for proactive security measures remain significant. Confidential computing addresses these challenges by reducing trust dependencies and ensuring the protection of sensitive data, even from cloud providers [528]. Integrating CHERI into cloud environments represents a promising avenue for addressing critical security challenges in modern cloud infrastructure, including virtual machines (VMs), containers, and hypervisors. In containerized cloud environments, CHERI capabilities extend to ensuring process-level isolation, crucial for preventing lateral movement between containers. Moreover, as cloud-native applications rely on microservices, CHERI can restrict each service's access to only necessary resources, minimizing the impact of potential exploits.

CAP-VM approaches (Section 5.5.5) align with and build upon these directions. A recent work [529] leverages CAP-VM to explore the potential of CHERI to compartmentalize a network stack (typically composed of an Ethernet driver, a Transmission Control Protocol / Internet Protocol (TCP/IP) library, and an application) while evaluating the trade-offs between security and performance degradation of varying levels of granularity on the *Arm Morello* platform. Future work should invest in adapting existing cloud software and frameworks to leverage CHERI protections, either by recompiling or by making compatibility adjustments. Cloud-based environments also tend to use TEEs, such as Intel SGX enclaves and AWS Amazon Nitro card [530], to guarantee security between trusted and untrusted execution. The Cloud Attestables on Morello Boards (CAMB) [531] project is using the *Arm Morello* prototype to demonstrate the potential of CHERI-enabled hardware for securing an execution environment.

5.6.2 CHERI for Mixed Criticality Systems

MCSs are prevalent in domains like automotive, aerospace, and robotics, where safety-critical operations coexist with less critical or non-critical functionalities. Ensuring security, reliability, predictability, and performance in such heterogeneous environments poses significant challenges, including the need to isolate critical components from potential faults or attacks in non-critical components. CHERI technology can facilitate high-level security for mixed-criticality systems, without significantly impacting their performance.

In the automotive domain, the AutoCHERI project [532] (a consortium led by Beam Connectivity) investigates the trade-offs of using CHERI technology in the automotive industry. Specifically, they: (i) study the possible use-cases for CHERI technology (e.g., teleoperation, OTA software updates), (ii) implement a Telematics Control Unit (TCU) for cloud services; and (iii) study the viability of CHERI-enabled products for the automotive industry. In a recent work, the *Arm Morello* board was explored to address critical

cyber-security risks in automotive systems [533]. Using the International Organization for Standardization (ISO)/Society of Automotive Engineers (SAE) 21434 framework and Threat Analysis and Risk Assessment (TARA), the study identified potential threats and defined security requirements. Key outcomes included secure TCU design, enhanced OTA update protection, and reduced risks for memory-related vulnerabilities. However, challenges such as long technology adoption cycles, complex supply chains, and certification requirements persist. Another study implemented a CHERI-enabled Intrusion Detection System (IDS) for automotive networks, showing CHERI applicability in this field [534].

Automotive is not the sole MCS-based domain where CHERI has attracted attention. At the RISC-V Summit Europe 2024, Ferraro et al. [535] presented an ongoing deployment of a CHERI-based platform for drone environments equipped with the CVA6 CHERI, highlighting innovative developments in the field of drone security and safety. By integrating CHERI, the platform ensures enhanced protection against vulnerabilities that are common in complex drone operations. The research extends PX4/NuttX [536] to separate drone functionalities into secure CHERI compartments. Real-time and security capabilities of the platform (with and without a hypervisor) are also being evaluated.

5.7 CHERI Future Directions

This section explores future directions for CHERI, focusing on the integration of capability-based protections for heterogeneous computing, large-scale applications, I/O or platform level, hypervisors, TEEs, and side-channel attacks.

5.7.1 CHERI for Heterogeneous Computing

Heterogeneous Computing architectures have become prominent in modern systems. These systems typically integrate various processing units, such as CPUs, GPUs, FPGAs, Tensor Processing Units (TPUs), and NPUs to enhance performance for critical tasks and improve energy efficiency. Such heterogeneity supports High-Performance Computing (HPC), supercomputing, and datacenter infrastructures, where specialized hardware units offload compute-intensive workloads, including scientific simulations, AI model training and execution, and real-time data analytics.

Heterogeneous systems often depend on software stacks developed in memory-unsafe languages like C and C++, notably within the GPU ecosystem. For example, NVIDIA's CUDA [537] and OpenCL [538] are general-purpose programming frameworks for GPUs based on C/C++, which are susceptible to various memory safety vulnerabilities, such as buffer overflows and use-after-free errors. Researchers have shown that GPU and other types of accelerators are susceptible to memory safety vulnerabilities [539–542], revealing a significant attack surface in contemporary heterogeneous systems.

To tackle these challenges, CHERI presents a promising framework for implementing fine-grained memory protection in heterogeneous computing environments. Originally designed for CPUs, CHERI's capability-based memory model has gained increasing interest for applications in heterogeneous settings.

Recent work by Naylor et al. [543] explores the feasibility of incorporating CHERI capabilities into a GPU architecture known as CHERI-Single Instruction Multiple Threads (SIMT), by adapting CHERI to the SIMT model utilized in GPUs. This adaptation effectively enforces spatial memory safety across threads while maintaining minimal performance overhead. Future work will study the integration of CHERI technology in commodity heterogeneous stacks.

5.7.2 CHERI for Large-Scale Applications

Ongoing work in the CHERI community is exploring the integration of CHERI for large-scale applications, with early reports demonstrating progress in both web services [544] and desktop environments [545]. These reports underline the benefit of vulnerability mitigations in such environments. CHERI enhances spatial and temporal memory safety and promotes strong compartmentalization, limiting the potential damage from attacks. This is especially useful in applications that execute multiple, potentially untrusted code components, such as open-source Graphical User Interface (GUI) software stacks and web server applications. On the desktop side, CHERI hardware-backed sandboxing mechanisms (e.g., Okular and KFileMetadata) help ensure that compromised plugins or extensions remain isolated and cannot affect the broader system. Additionally, everyday applications such as document editors, multimedia players, and email clients can compartmentalize their core functions thereby mitigating the risks associated with handling untrusted files or unexpected software bugs.

However, the adoption CHERI for mobile devices faces considerable challenges. This is primarily due to the highly concentrated nature of the market and the mature, feature-rich hardware ecosystems dominated by a small number of players. As of this writing, current mobile architectures prioritize power efficiency and compact design, leaving limited space for new features like CHERI. Despite these challenges, the memory safety benefits of CHERI could provide strong incentives in the future. To the best of our knowledge, only one work-in-progress [546] aims to provide an Android stack for the *Arm Morello* platform. Nonetheless, this research could help key players like Google, Apple, and Microsoft to continue to invest in CHERI and potentially pave the way for its broader adoption.

5.7.3 CHERI for I/O

On a SoC, attackers can circumvent CPU protections by using I/O devices with DMA capabilities that are not subject to the same enforcement as in a CHERI CPU. In capability systems, DMA-based attacks can expose security holes that can corrupt capabilities and bypass the CHERI protection model at the CPU level. Redefining or extending the existing memory protection mechanism for I/O devices is necessary to achieve full protection and trustworthiness in a CHERI-based system.

However, the literature on defining protection mechanisms for I/O devices in CHERI systems is relatively unexplored. To date, the CHERI specification has followed a conservative approach, stripping the

tag from the data for I/O memory accesses. This presents an opportunity for further research and innovation in the field. Markettos et al. [547] argued that CHERI can enhance security with capability-based access control for I/O new components and in-place replacements for this traditional protection mechanism, such as the RISC-V IOMMU and IOPMP. For example, the IOMMU can be simplified by offering single-level translation and protection based on capabilities. In addition, CHERI could replace the IOPMP, providing flexibility (the mapping of objects to IOPMP registers can be removed), and avoiding potential problems due to partial configurations. Moreover, there are several system-level isolation technologies, such as RISC-V WorldGuard [65], that can be integrated with CHERI technology or even be replaced by it. Further analysis is needed to assess the impact of CHERI on these technologies.

5.7.4 CHERI for Hypervisors

Hypervisors enable virtualization by running multiple operating systems or applications on shared hardware with strong isolation guarantees, leading to more effective use of the hardware resources. Hypervisors are classified as Type-1 (baremetal) and Type-2 (hosted). Type-1 hypervisors, such as Xen [115] and Bao [118], run directly on the hardware, managing virtual machines without a host operating system. Type-2 hypervisors, like *bhyve* [548] and Kernel-based Virtual Machine (KVM) [549], run on top of a host OS, such as FreeBSD or Linux, relying on its services for virtualization.

Integrating CHERI into hypervisors presents several technical challenges. For example, (i) hypervisors must adapt their virtual memory subsystems to correctly handle capabilities alongside traditional page tables, and, (ii) adapting hypervisor source code to use CHERI capabilities requires rewriting, especially for optimized code executing unsafe memory accesses, potentially leading to performance degradation. Despite these challenges, the authors of the CHERI-TrEE work have implemented a trusted hypervisor on *Arm Morello* to mimic and evaluate the CHERI-TrEE operations [550]. The hypervisor runs in EL2, pausing the core execution at EL1 to perform similar mechanisms of CHERI-TrEE core. Despite not providing many typical hypervisor primitives, the implementation shows the feasibility of a TEE that leverages CHERI capabilities and a possible use of the hypervisor feature of the Arm Morello platform.

There is ongoing work to port the *bhyve* hypervisor to CHERI [551] running atop FreeBSD. The work aims at showcasing how existing virtualization systems can incrementally adopt CHERI capability-based protections on CheriBSD for *Arm Morello*. However, at the time of this writing, bhyve is still an experimental feature. Additional works include the integration of *Bao* [118] in the CHERI ecosystem using the RISC-V CVA6 with Hypervisor and CHERI extensions [443]. By replacing traditional memory management with CHERI capabilities, Bao can achieve fine-grained memory safety and stronger partitioning guarantees.

5.7.5 CHERI for TEE

TEEs are isolated areas of a processor that secure sensitive applications (e.g., cryptographic operations) from the rest of the system. A typical implementation strategy for TEEs is enclaving specific regions of

code and data, ensuring that even privileged software, such as the operating system or hypervisor, cannot access or tamper with its contents. Examples of enclaving technologies include Intel SGX [59], AMD SEV [376], and ARM TrustZone [58].

As a future direction, TEE will be a strong use case for the CHERI technology. Van Strydonck et al. [550] have shown that with minor modifications to the ISA, CHERI technology provides a clean way to define enclaves in an embedded system without an MMU. Another project, CAP-TEE [484], is studying how the CHERI technology can be used to mitigate state-of-the-art attacks on TEE environments (e.g., memory corruption and control-flow hijack [552]). For instance, they implemented a CHERI-enabled memory encryption engine that extends the CHERI ISA with encryption semantics to create and invoke enclaves [553]. Other studies (e.g., Capstone [554]) evaluate capability-based protection for TEEs. These studies suggest that the capability-based compartmentalization provided by CHERI can improve general TEE deployments. Compared to traditional TEE approaches, CHERI adds deterministic spatial memory safety and fine-grained compartmentalization. For instance, current enclave technologies for RISC-V (e.g., Keystone [60] or CURE [382]) rely on the RISC-V PMP unit to isolate memory. The number of PMP entries is typically scarce, limiting the maximum number of enclaves. Nevertheless, these studies are limited, and further research on CHERI for TEEs is needed.

5.7.6 CHERI for Side-Channel Attacks

Side-channel attacks are a significant concern in security research. These attacks were highlighted by Spectre [326] and Meltdown [555] vulnerabilities. Modern CPUs utilize speculative execution to enhance instruction parallelism and performance, but this can leave traces in microarchitectural components that are exploitable through timing channels [556]. These attacks exploit microarchitectural speculation to breach security boundaries and extract sensitive data. While the security model of CHERI was not explicitly designed to increase side-channel resilience, it may inadvertently create new vulnerabilities [557] while also offering potential mitigations [557, 558].

Several studies have already examined the impact of CHERI technology on side-channel attacks. One proposed solution involves using a CID that tags the microarchitectural state with a unique identifier, marking the current context and mitigating leaks between CHERI compartments [559]. This mechanism has been tested on the *Arm Morello* prototype (see Section 5.4.4.3). Fuchs et al. [395] have developed a bare-metal test suite that replicates various transient-execution attacks in RISC-V and CHERI RISC-V processors. These tests include adaptations of the Spectre and Meltdown attacks for the CHERI ISA, using the CHERI-Toooba processor as a case study. While they demonstrated that CHERI is resilient to typical Meltdown attacks, they found that CHERI does not prevent Spectre attacks that exploit speculative mechanisms. Later, Fuchs et al. [557] defined CHERI ISA-level safe speculation hardware-software contracts [558] that limit and enforce boundaries for speculation mechanisms at the microarchitectural implementation while allowing the software to reason about the execution of programs. The study also identified a new type of Meltdown attack, specifically targeting the CHERI-Toooba implementation, called Meltdown-CF [557].

Henes et al. [560] conducted a preliminary study on the transient execution of Spectre-like attacks on the *Arm Morello* platform, showing that this platform is vulnerable to attacks that rely on capability speculation, such as Spectre-BTB. Future research will continue to explore and enhance the CHERI technology to mitigate a broader class of side-channel attacks (e.g., power analysis attacks), while trying to identify novel vectors of attacks that may be introduced by the CHERI capabilities.

Despite a growing body of work, several important questions remain open. A key open issue is how CHERI capabilities impact timing side-channels in microarchitectural shared resources such as caches, TLBs, and interconnects. Because capabilities are larger than conventional pointers (typically by a factor of two), they impose a substantially greater cache footprint and generate increased memory traffic, which we speculate may amplify timing-based side channels. Furthermore, although mechanisms such as CIDs are designed to partition microarchitectural state at the CPU level and thereby reduce interference between protection domains, the degree to which they actually prevent unintended information flow between CHERI compartments remains insufficiently understood. Clarifying whether residual sharing of microarchitectural resources can still facilitate cross-compartment leakage and how CID-based partitioning composes with speculative execution, caching policies, and interconnect behavior, constitutes a significant open research problem. Consequently, future work should focus on providing additional microarchitectural mechanisms to prevent timing-based side-channel attacks on CHERI-based platforms.

5.8 Conclusion

This article presented a comprehensive survey of CHERI technology and highlighted its potential in enhancing system security through hardware-enforced memory safety. CHERI provides fine-grained memory protection and robust compartmentalization mechanisms, offering a significant improvement in securing modern systems against most common vulnerabilities. The growing support for CHERI from both academia and industry, coupled with initiative such as UK government's DSbD program, have demonstrated CHERI's applicability across domains ranging from traditional computing to IoT and real-time systems. From the literature and practical implementations reviewed, it is evident that CHERI has achieved significant progress over the past decade. At the hardware level, platforms like *Arm Morello* and *Sonata* have provided experimental insights into CHERI's benefits in terms of security and performance. These advancements are complemented by substantial efforts to integrate CHERI into software ecosystems, including operating systems, compilers, and applications. This article aims to help researchers, developers, and system architects understand the current state of the art of CHERI technology and inspire further innovation. From our study, we observe opportunities for more exploration of CHERI technology, particularly in improving its adoption in existing software stacks and in enhancing the ISA by addressing deployment challenges in large-scale and resource-constrained environments.

6

CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration

This chapter was published as:
Bruno Sá, Luca Valente, José Martins, Davide Rossi, Luca Benini, and Sandro Pinto. "CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration". In IEEE Transactions on Very Large Scale Integration (VLSI) Systems

Abstract

Virtualization is a key technology used in a wide range of applications, from cloud computing to embedded systems. Over the last few years, mainstream computer architectures were extended with hardware virtualization support, giving rise to a set of virtualization technologies (e.g., Intel VT, Arm VE) that are now proliferating in modern processors and SoCs. In this article, we describe our work on hardware virtualization support in the RISC-V CVA6 core. Our contribution is multifold and encompasses architecture, microarchitecture, and design space exploration. In particular, we highlight the design of a set of microarchitectural enhancements (i.e., GTLB, L2 TLB) to alleviate the virtualization performance overhead. We also perform a DSE and accompanying post-layout simulations (based on 22nm FDX technology) to assess PPA. Further, we map design variants on an FPGA platform (Genesys 2) to assess the functional performance-area trade-off. Based on the DSE, we select an optimal design point for the CVA6 with hardware virtualization support. For this optimal hardware configuration, we collected functional performance results by running the MiBench benchmark on Linux atop Bao hypervisor for a single-core configuration. We observed a performance speedup of up to 16% (approx. 12.5% on average) compared with virtualization-aware, non-optimized design at cost of only a 0.78% in area and 0.33% in power. Finally, all work described in this article is publicly available and open-sourced for the community to further evaluate additional design configurations and software stacks.

6.1 Introduction

Virtualization is a technological enabler used on a wide range of applications, ranging from cloud computing and servers to mobile devices and embedded systems [118]. As a fundamental cornerstone of cloud computing, virtualization provides numerous advantages for workload management, data protection, and cost-/power-effectiveness [9]. On the other side of the spectrum, the embedded and safety-critical systems industry has resorted to virtualization as a fundamental approach to address the market pressure to minimize SWaP-C, while guaranteeing temporal and spatial isolation for certification (e.g., ISO26262) [7, 8, 561]. Due to the proliferation of virtualization across multiple industries and use cases, prominent players in the silicon industry started to introduce hardware virtualization support in mainstream computing architectures (e.g., Intel Virtualization Technology, Arm Virtualization Extensions, respectively) [54, 55].

Recent advances in computing architectures have brought to light a novel ISA named RISC-V [168]. RISC-V has recently reached the mark of 10+ billion shipped cores [562]. It distinguishes itself from the classical mainstream by providing a free and open standard ISA, featuring a modular and highly customizable extension scheme that allows it to scale from small microcontrollers up to supercomputers [563–569]. The RISC-V privileged architecture provides hardware support for virtualization by defining the Hypervisor extension [570], ratified in Q4 2021.

Despite the Hypervisor extension ratification, as of this writing, there is no RISC-V silicon with this

extension on the market[1]. There are open-source hypervisors with upstream support for the Hypervisor extension, i.e., Bao [118], Xvisor [571], KVM [572], and seL4 [573] (and work in progress in Xen [574] and Jailhouse [575]). However, to the best of our knowledge, there are just a few hardware implementations deployed on FPGA, which include the Rocket chip [56] and NOEL-V [576] (and soon SHAKTI and Chromite [577]). Notwithstanding, no existing work has (i) focused on understanding and enhancing the microarchitecture for virtualization and (ii) performed a DSE and PPA analysis.

This work describes the architectural and microarchitectural support for virtualization in an open-source RISC-V CVA6-based [567] (64-bit) SoC. At the architectural level, the implementation is compliant with the Hypervisor extension (v1.0) [177] and includes the implementation of the RISC-V timer (Sstc) extension [578] as well. At the microarchitectural level, we modified the vanilla CVA6 microarchitecture to support the Hypervisor extension and proposed a set of additional extensions/enhancements to reduce the hardware virtualization overhead: (i) a dedicated second stage TLB coupled to the PTW (i.e., GTLB in our lingo), and (ii) a second level TLB (L2 TLB). We also present and discuss a comprehensive design space exploration on the microarchitecture. We first evaluate 23 (out of 288) hardware designs deployed on FPGA (Genesys 2) and assess the impact on functional performance (execution cycles) and hardware. Then, we elect 7 designs and analyze them in depth with post-layout simulations of implementations in 22nm FDX technology.

We ran the MiBench (automotive subset) benchmarks for the DSE evaluation to assess functional performance. The virtualization-aware, non-optimized CVA6 implementation served as our baseline configuration. The software setup encompassed a single Linux VM running atop Bao hypervisor for a single-core design. We measured the performance speedup of the hosted Linux VM relative to the baseline configuration. Results from the DSE exploration demonstrated that the proposed microarchitectural extensions could achieve a functional performance speedup of up to 19% (e.g., for the *susanc* small benchmark); however, in some cases at the cost of a non-negligible increase in area and power. Thus, results from the PPA analysis show that: (i) the Sstc extension has negligible impact on power and area; (ii) the GTLB increases the overall area by less than 1%; and (iii) the L2 TLB introduces a non-negligible 8% increase in area in some configurations. As a representative, well-balanced configuration, we selected the CVA6 design with Sstc support and a GTLB with 8 entries. For this specific hardware configuration, we observed a performance speedup of up to 16% (approx. 12.5% on average) at the cost of 0.78% in area and 0.33% in power. To the best of our knowledge, this paper reports the first public work on a complete DSE evaluation and PPA analysis for a virtualization-enhanced RISC-V core.

To summarize, with this work, we make the following contributions. Firstly, we provide hardware virtualization support for the CVA6 core, which was completely absent in the vanilla implementation. In particular, we implement the RISC-V Hypervisor extension (v1.0) and design a set of (virtualization-oriented) microarchitectural enhancements to the Nested Memory Management Unit (Section 6.3). To the best of our knowledge, no work or study describes and discusses microarchitectural extensions to improve the

[1]SiFive, Ventana, and StarFive have announced RISC-V CPU designs with Hypervisor extension support, but we are not aware of any silicon available on the market yet.

Name	Definition
Abbreviations	
PTW	Page Table Walker. The PTW is responsible for traversing the page table and converting a virtual address into a physical address.
DSE	Design Space Exploration. Functional evaluation carried out for multiple FPGA design configurations.
PPA	Power, Performance, Area. Post-layout silicon power, performance, and area analysis.
VM	Virtual Machine. The execution environment (e.g., OS or application) running atop a hypervisor.
Nested-MMU	Nested Memory Management Unit. Using two translation stages to convert a guest's virtual address into a host's physical address. The first is controlled by the guest OS and the second by the hypervisor.
TLB	Translation Lookaside Buffer. Microarchitectural element used to cache translations from virtual addresses to physical addresses.
VS-Mode	Virtual Supervisor Mode. Privilege mode in RISC-V, where a guest OS runs when virtualized.
HS-Mode	Hypervisor-extended Supervisor Mode. Privilege mode in RISC-V, where the hypervisor runs.
GTLB	G-stage Translation Lookaside Buffer. TLB that stores G-stage only translation, i.e., guest physical addresses into host physical addresses.
RISC-V Terminologies	
CSR	Control Status Registers. RISC-V CPU control registers.
VS-Stage	RISC-V terminology for a first translation stage. Converts guest virtual addresses into guest physical addresses.
G-Stage	RISC-V terminology for a second translation stage. Converts guest physical addresses into host physical addresses.
ASID	Address Space Identifier. Processes have a unique identifier used to tag translations on the MMU (e.g., on TLB entries).
VMID	Virtual Machine Space Identifier. Guest virtual machines have a unique identifier used to tag translations on the MMU.
PTE	Page Table Entry. Specific entry of a page table used to convert virtual addresses into physical addresses.
RISC-V Control Status Registers and Instructions	
stimecmp	Supervisor Time Comparator Register. Generates timer interrupts in HS/S-mode. Only available if the RISC-V Sstc extension is implemented.
vsatp	Virtual Supervisor Guest Address Translation and Protection Register. Controls the VS-stage address translation and protection.
vstimecmp	Virtual Supervisor Time Comparator Register. Generates timer interrupts at VS-mode. Only available if the RISC-V Sstc extension is implemented.
hgeie and hgeip	Hypervisor Guest External Enable and Pending Registers. Deliver interrupts directly to the guest if the interrupt controller supports virtualization.
henvcfg	Hypervisor Environment Configuration Register. Controls hypervisor access (enabled or disabled) to optional platform-specific extensions (e.g., Sstc extension).
hgatp	Hypervisor Guest Address Translation and Protection Register. G-stage root table pointer and respective translation-specific configuration fields.
menvcfg	Machine Environment Configuration Register. Controls firmware access (enabled or disabled) to optional platform-specific extensions (e.g., Sstc extension).
hfence	Hypervisor Fence Instruction. Flush cached translations on TLBs and other related structures. The hfence.vvma flushes VS-Stage and G-Stage translations. The hfence.gvma flushes only G-Stage translations.

Table 17: Table of abbreviations, RISC-V terminologies and RISC-V Control Status Registers and Instructions.

hardware virtualization support in a RISC-V core. Second, we perform a DSE encompassing dozens of design configurations. This DSE includes trade-offs on parameters from three different microarchitectural components (L1 TLB, GTLB, L2 TLB) and respective impact on functional performance and hardware costs (Section 6.4). Finally, we conduct post-layout simulations on a few elected design configurations to assess a PPA analysis (Section 6.5). All contributions described in this manuscript are open source [2] and available to the RISC-V community to foster collaboration and enable contributions with further extensions, optimizations, and testing/verification activities. The Hypervisor extension is now being upstreamed to the CVA6 main repository[3]. Our goal is to democratize virtualization for the next-generation CVA6-based SoCs.

6.2 Background

This section covers the background related to RISC-V technology (ISA, extensions, and cores) and virtualization. To improve readability, we also provide a table of abbreviations and terminologies used throughout the article (see Table 17).

[2] https://github.com/minho-pulp/cva6/

[3] https://github.com/minho-pulp/cva6/tree/feat/hyp-upstream

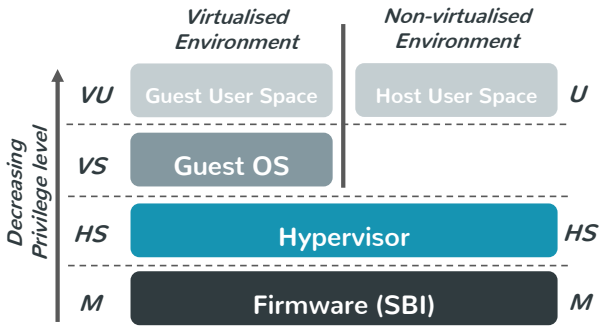


Figure 19: RISC-V privilege levels: machine (M), hypervisor-extended supervisor (HS), virtual supervisor (VS), and virtual user (VU).

6.2.1 Virtualization Technology

Virtualization is the de facto technology that consolidates and isolates multiple non-related software stacks onto the same hardware platform by partitioning and multiplexing hardware resources (e.g., CPUs, memory) between multiple virtual machines. The VMM or hypervisor is the software layer that implements virtualization. The software executing in the VM is referred to as a guest, typically an OS, i.e., guest OS.

6.2.2 RISC-V Privileged Specification

The RISC-V privileged ISA [570] divides its execution model into 3 privilege levels: (i) *machine mode* (M-mode) is the most privileged level, hosting the firmware which implements the SBI (e.g., OpenSBI); (ii) *supervisor mode* (S-Mode) runs Unix-like OSs that require virtual memory management; (iii) *user mode* (U-Mode) executes userland applications. The modularity offered by the RISC-V ISA seamlessly allows for implementations at distinct design points, ranging from small embedded platforms with just M-mode support to fully-blown server-class systems with M/S/U.

6.2.3 RISC-V Virtualization

Unlike other mainstream ISAs, the RISC-V privileged architecture was designed from the initial conception to be classically virtualizable [114]. So although the ISA, per se, allows the straightforward implementation of hypervisors resorting, for example, to classic virtualization techniques (e.g., trap-and-emulation and shadow page tables), it is well understood that such techniques incur a prohibitive performance penalty and cannot cope with current embedded real-time virtualization requirements (e.g., interrupt latency) [56]. Thus, to increase virtualization efficiency, the RISC-V privileged architecture specification introduced hardware support for virtualization through the (optional) *Hypervisor* extension [177]. The following paragraphs provide a high-level overview of the RISC-V Hypervisor extension specification.

Privilege Levels. As depicted in Figure 19, the RISC-V Hypervisor extension execution model follows an orthogonal design where the *supervisor mode* (S-mode) is modified to a *hypervisor-extended supervisor*

mode (HS-mode) well-suited to host both Type-1 or Type-2 hypervisors [4]. Additionally, two new privileged modes are added and can be leveraged to run the guest OS at *virtual supervisor mode* (VS-mode) and *virtual user mode* (VU-mode).

Two-stage Address Translation. The Hypervisor extension also defines a second translation stage (*G-stage* in RISC-V lingo) to virtualize the guest memory by translating guest-physical addresses into host-physical addresses. The HS-mode operates like S-mode but with additional hypervisor registers and instructions to control the VM execution and *G-stage* translation. For instance, the *hgatp* register holds the *G-stage* root table pointer and translation-specific configuration fields.

Hypervisor CSRs. Each VM running in VS-mode has its own CSRs that are shadow copies of the S-mode CSRs. These registers can be used to determine the guest execution state and perform VM switches. To control the virtualization state, a specific flag called *virtualization mode* (V bit) is used. When V=1, the guest is executing in VS-mode or VU-mode, normal S-mode CSRs accesses actually access the VS-mode CSRs, and the *G-stage* translation is active. Otherwise, if V=0, normal S-mode CSRs are active, and the *G-stage* is disabled. To ease guest-related exception trap handling, there are guest-specific traps, e.g., guest page faults, VS-level illegal exceptions, and VS-level *ecalls* (a.k.a. hypercalls).

Hypervisor Instructions. The Hypervisor extension defines a set of hypervisor-related instructions to increase the virtualization efficiency. These instructions encompass: (i) hypervisor load/store instructions used to access guest memory with the exact translation and permission rights (RWX) as the guest; and (ii) hypervisor fence instructions to flush guest-related cached translation structures (e.g., TLBs) during context switching operations.

6.2.4 Nested Memory Management Unit (Nested-MMU)

The MMU is a hardware component responsible for translating virtual memory references to physical ones while enforcing memory access permissions. The OS controls the MMU by assigning virtual address space to each process and managing the MMU translation structures to translate virtual addresses into physical addresses correctly. On a virtualized system, the MMU provides another layer of translation and protection controlled by the hypervisor. In this case, the MMU can translate from guest-virtual addresses to guest-physical addresses and from guest-physical addresses into host-physical addresses. This new feature is referred to as nested-MMU. The RISC-V ISA supports the nested-MMU through a new stage of translation that converts guest-physical addresses into host-physical addresses, denoted G-stage. The guest VM takes control over the first stage of translation (VS-stage in RISC-V lingo), while the hypervisor assumes control over the second one (G-stage). Originally, the RISC-V privileged specification defines that a virtual address is converted into a physical address by traversing a multi-level radix-tree table using one of four different topologies: (i) *Sv32* for 32-bit virtual address spaces with a 2-level hierarchy tree; (ii)

[4]The main difference between Type-1 (or baremetal) and Type-2 (or hosted) hypervisors is that a type-1 hypervisor runs directly on the hardware (e.g., Xen) while a type-2 hypervisor runs atop an operating system (e.g., VMware).

Sv39 for 39-bit virtual address spaces with a 3-level tree; (iii) *Sv48* for 48-bit VAS and 4-level tree; and (iv) *Sv57* for 57-bit virtual address spaces and 5-level tree. Each level holds a pointer to the following table (non-leaf entry) or the final translation (leaf entry). This pointer and permissions are stored in a 64-bit (RV64) or 32-bit (RV32) width PTE. Note that RISC-V splits the virtual address into 4KiB page sizes, but since each level can either be a leaf or non-leaf, it supports superpages to reduce the TLB pressure, e.g., Sv39 supports 4KiB, 2MiB, and 1GiB page sizes.

6.2.5 RISC-V "stimecmp/vstimecmp" Extension (Sstc)

The RISC-V Sstc extension [578] aims to enhance supervisor mode with its timer interrupt facility, thus eliminating the large overheads for emulating S/HS-mode timers and timer interrupt generation up in M-mode. The Sstc extension also adds a similar facility to the Hypervisor extension for VS-mode. To enable direct control over timer interrupts in the HS/S-mode and VS-mode, the Sstc encompasses two additional comparators: (i) *stimecmp* register to generate S/HS-mode timer interrupts and (ii) *vstimecmp* register to generate VS-mode timer interrupts. Whenever the value of the system clock *time* counter register is greater than the value on the comparators, a timer interrupt is generated. For a complete overview of the RISC-V timer architecture and a discussion on why the classic RISC-V timer specification incurs a significant performance penalty, we refer the interested reader to [56].

6.2.6 CVA6

CVA6 is an application-class RISC-V core that implements both the RV64 and RV32 versions of RISC-V ISA [567]. The core fully supports the 3 privilege execution modes M/S/U-modes and provides hardware support for memory virtualization by implementing a MMU, making it suitable for running a fully-fledged OS such as Linux. Recent additions also include an energy-efficient Vector Unit co-processor [179]. Internally, the CVA6 microarchitecture encompasses a 6-stage pipeline, single-issue, with an out-of-order execution stage and 8 PMP entries. The MMU has separate TLBs for data and instructions and the PTW implements the *Sv39* and *Sv32* translation modes as defined by the privileged specification [177].

6.3 CVA6 "From Scratch" Virtualization Support

This section reports our work in adding hardware support for virtualization in the CVA6 core (compliant with the RISC-V Hypervisor extension v1.0). To the best of our knowledge, this work is the first initiative to add hardware virtualization support in the CVA6 core. The following section divides into seven logical subsections: (i) Subsections 6.3.1, 6.3.2, 6.3.3 and 6.3.4 describe the modifications/additions to CVA6 microarchitecture enabling the Hypervisor extension support, specifically to the core registers and the MMU subsystem; (ii) Subsection 6.3.5 and 6.3.6 presents the microarchitectural enhancements designed in the MMU subsystem to increase virtualization efficiency; and (iii) Subsection 6.3.7 describes how

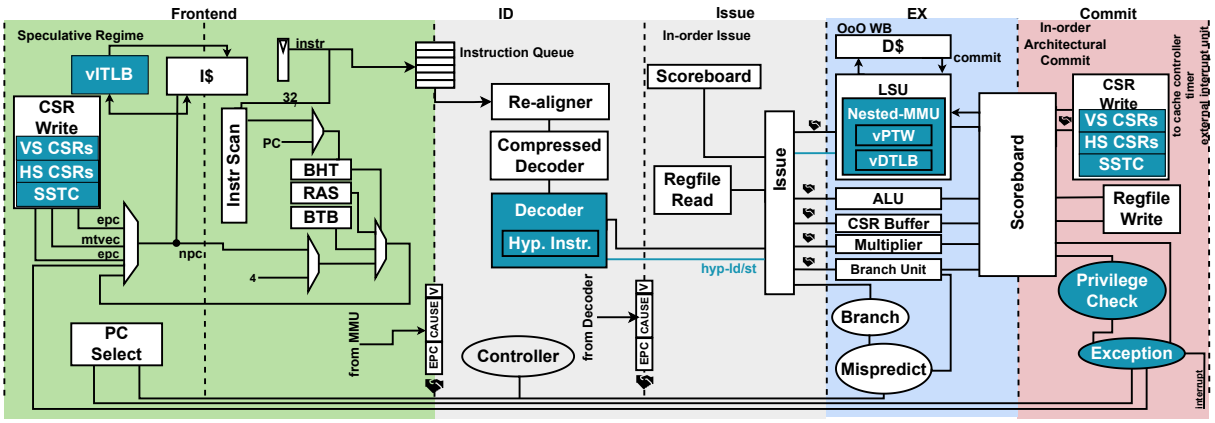


Figure 20: CVA6 core microarchitecture featuring the RISC-V hypervisor extension. Major microarchitectural changes to the core functional blocks (e.g., Decoder, PTW, TLBs, and CSRs) are highlighted in blue. Adapted from [567].

we extend the CVA6 timer with the standard RISC-V Sstc extension to improve the timer management infrastructure. Figure 20 illustrates a high-level overview of the modifications/additions performed in the CVA6 microarchitecture.

6.3.1 Hypervisor and Virtual Supervisor Execution Modes

As previously described (refer to Section 6.2.3), the Hypervisor extension specification extends the S-mode into the HS-mode and adds two extra orthogonal execution modes, denoted VS-mode and VU-mode. To add support for these new execution modes, we have extended/modified some of the CVA6 core functional blocks, in particular, the *CSR* and *Decode* modules. As illustrated by Figure 20, the hardware virtualization architecture logic encompasses five building blocks: (i) VS-mode and HS-mode CSR access logic and permission checks; (ii) exception and interrupt triggering and delegation; (iii) trap entry and exit; (iv) hypervisor instruction decoding and execution; and (v) nested-MMU translation logic. The *CSR module* was extended to implement the first three building blocks that comprise the hardware virtualization logic, specifically: (i) HS-mode and VS-mode CSRs access logic (read/write operations); (ii) HS and VS execution mode trap entry and return logic; and (iii) a fraction of the exception/interrupt triggering and delegation logic from M-mode to HS-mode and/or to VS/VU-mode (e.g., reading/writing to *vsatp* CSR triggers an exception in VS-mode when not allowed by the hypervisor). The *Decode* module was modified to implement hypervisor instructions decoding (e.g., hypervisor load/store instructions and memory-management fence instructions) and to detect and trigger access exceptions for VS-mode-related instructions.

We refer readers to Table 18, which presents a summary of the features that were fully and partially implemented. We have implemented all mandatory features of the ratified specification (v1.0); however, we still left some optional features as partially implemented due to the dependency on upcoming or newer extensions. For example, *henvcfg* bits depend on *Zicbom* [579] (cache block management operations);

hgeie and *hgeip* depend on the AIA [580]; and *hgatp* depends on virtual address spaces not currently supported in the vanilla CVA6.

CSRs	hstatus/mstatus	●
	hideleg/hedeleg/mideleg	●
	hvip/hip/hie/mip/mie	●
	hgeip/hgeie	◐
	hcounteren	●
	htimedelta	●
	henvcfg	◐
	mtval2/htval	●
	mtinst/htinst	●
	hgatp	◐
	vsstatus/vsip/vsie/vstvec/vsscratch vsepc/vscause/vstval/vsatp	●
Instructions	hlv/hlvx/hsv	●
	hfence.vvma/gvma	●
Exceptions & Interrupts	Environment call from VS-mode	●
	Instruction/Load/Store guest-page fault	●
	Virtual instruction	●
	Virtual Supervisor sw/timer/external interrupts	●
	Supervisor guest external interrupt	●

Table 18: Hypervisor extension features implemented in the CVA6 core: ● fully-implemented; ◐ partially implemented.

6.3.2 Hypervisor Load/Store Instructions

The hypervisor load/store instructions (i.e., *HLV*, *HSV*, and *HLVX*) provide a mechanism for the hypervisor to access the guest memory while subject to the the same translation and permission checks as VS-mode or VU-mode. These instructions change the translation settings at instruction granularity, forcing a complete swap of privilege level and translation applied at every hypervisor load/store instruction execution. The implementation encompasses the addition of a signal (identified as *hyp_ld/st* in Figure 20) to the CVA6 pipeline that travels from the decoding to the load/store unit in the execute stage. This signal is then fed into the MMU that performs (i) all necessary context switches (i.e., enables the *hgatp* and *vsatp* CSRs), (ii) enables the virtualization mode, and (iii) changes execution mode.

6.3.3 Nested Page-Table Walker

One of the major functional blocks of an MMU is the PTW. Fundamentally, the PTW is responsible for partitioning a virtual address according to the specific topology and scheme (e.g., *Sv39*) and then translating it into a physical address using the page-tables structures. The Hypervisor extension specifies a new translation stage (G-stage) used to translate guest-physical addresses into host-physical addresses. Our implementation supports *Bare* translation mode (no G-stage) and *Sv39x4*, which defines a 41-bit width maximum guest physical address space. As of this writing, the CVA6 only supported the *Sv39* scheme for the 64-bit core, so we extended the implementation to support the *Sv39x4*.

We extended the existing finite state machine (FSM) used to translate virtual addresses to physical addresses and added only a new control state to keep track of the current translation stage and assist the context switching between VS-Stage and G-Stage translations. With the G-stage in situ, it is mandatory to translate: (i) the resulting guest-physical address from the VS-Stage translation and (ii) all memory accesses made during the VS-Stage translation walk. To accomplish that, we identify three stages of translation that can occur during a PTW iteration: (i) *VS-Stage* - the PTW current state is translating a guest-virtual address into a guest-physical address; (ii) *G-Stage Intermed* - the PTW current state is translating memory accesses made from the VS-Stage during the walk into host-physical addresses; and (iii) *G-Stage Final* - the PTW current state is translating the final output address from VS-Stage into a host-physical address. It is worth noting that if the MMU is configured in Bare mode for the G-stage translation, we perform a standard S/VS-Stage translation. Once the nested walk completes, the PTW updates the TLB with the final PTE from VS-stage and G-stage, alongside the current ASID and the Virtual-Machine Identifier (VMID). VMID tags each translation to a specific VM. One implemented optimization consists of storing the translation page size (i.e., 4KiB, 2MiB, and 1GiB) for both VS- and G-stages into the same TLB entry and permissions access bits for each stage.

6.3.4 Virtualization-aware TLBs (vTLB)

The CVA6 MMU microarchitecture has two small, fully associative TLBs: data (L1 DTLB) and instructions (L1 ITLB). Both TLBs support a maximum of 16 entries and fully implement the flush instructions, i.e., *sfence*, including filtering by ASID and virtual address. To support nested translation, we modified the L1 DTLB and ITLB microarchitecture to support two translation stages, including access permissions and VMIDs. Each TLB entry holds both VS-Stage and G-Stage PTEs and respective permissions. The look-up logic is performed using the merged final translation size from both stages, i.e., if the VS-stage is a 4KiB and the G-stage is a 2MiB translation, the final translation would be a 4KiB. This is probably one of the major drawbacks of having both the VS-stage and G-stage stored together. For instance, hypervisors supporting superpages/hugepages use the 2MiB page size to optimize the MMU performance. Although this significantly reduces the PTW walk time, if the guest uses a 4KiB page size, the TLB look-up would not benefit from superpages since the translation would be stored as a 4KiB page size translation. A possible alternative would be having separate TLBs for each stage. We argue that this solution would have three

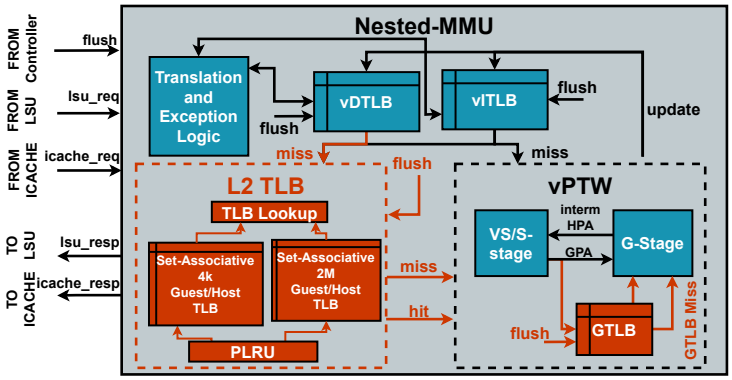


Figure 21: MMU microarchitectural enhancements: high-level overview. Additions to the original MMU design highlighted in orange.

major drawbacks: (i) negative impact on performance as it would be possible to search in G-Stage TLB after the VS-Stage TLB look-up (i.e., increase the TLB hit time penalty by a factor of 2); (ii) less hardware reuse, i.e., the G-Stage TLB is only used when a guest is running and G-stage is active; and (iii) more impact on the area and energy, the CVA6 TLBs are fully combinational circuits with a significant impact on the area, energy, and timing. For all the above reasons, we decided to keep the VS-stage and G-stage translations in a single TLB entry. Additionally, the TLB also supports VMID tags allowing hypervisors to perform a more efficient TLB management using per-VMID flushes and avoiding full TLB flush on a VM context switch. Finally, the TLB also allows flushes by guest physical address, i.e., hypervisor fence instructions (*hfence.vvma/gvma*) are fully supported.

6.3.5 Microarchitectural extension #1 - GTLB

A full nested page-table walk for the *Sv39* scheme can take up to 15 memory accesses, five times as many than a standard S-stage translation. This additional burden imposes a higher TLB miss penalty, resulting in a (measured) overhead of up to 13% on the functional performance (execution cycles) (comparing the baseline virtualization implementation with the vanilla CVA6). To mitigate this, we have extended the CVA6 microarchitecture with a GTLB in the nested-PTW module to store intermediate GPA to HPA translations, i.e., VS-Stage non-leaf PTE guest physical address to host physical addresses translation. Figure 21 illustrates the modifications required to integrate the GTLB in the MMU microarchitecture. The GTLB structure aims at accelerating VS-stage translation by skipping each nested translation during the VS-stage page table walk. Figure 22 presents a 4KiB page size translation process featuring a GTLB in the PTW. Without the GTLB, each time the guest forms the non-leaf physical address PTE pointer during the walk, it needs: (i) to translate it via G-stage, (ii) read the next level PTE value from memory, and (iii) resume the VS-stage translation. When using superpages (2MiB or 1GiB), there are fewer translation walks, reducing the performance penalty. With a simple hardware structure, it is possible to mitigate such overheads by keeping those G-stage translations cached while avoiding unnecessary cache pollution and nondeterministic memory accesses.

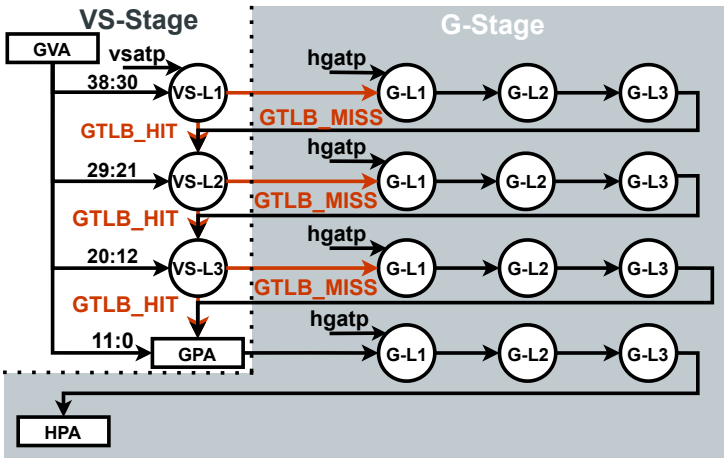


Figure 22: RISC-V nested-MMU walk example for the SV39x4 scheme.

GTLB microarchitecture. The GTLB follows a fully-associative design with support for all *Sv39* superpage sizes (4KiB, 2MiB, and 1GiB). Each entry is searched in parallel during the TLB look-up, and each translation can be stored on any TLB entry. The replacement policy evicts the least recently used entry using a pseudo-least recently used (PLRU) algorithm already implemented on the CVA6 L1 TLBs. The maximum number of entries that GTLB can hold simultaneously ranges from 8 to 16. The flush circuit fully supports hypervisor fence instructions (*HFENCE.GVMA*), including filtering flushed TLB entries by VMIDs and virtual address. It is worth mentioning that the GTLB implementation reused the already-in-place L1 TLB design and modified it to store only G-stage-related translations. We map TLB entries to flip-flops due to their reduced number.

6.3.6 Microarchitectural extension #2 - L2 TLB

Currently, the vanilla CVA6 MMU features only a small separate L1 instruction and data TLBs, shared by guests and the hypervisor. The TLB can merge VS and G stage translations in one single entry, where the stored translation page size must be the minimum of the two stages. This may improve hardware reuse and save some additional search cycles, as no separated TLBs for each stage are required. However, as explained in Subsection 6.3.4, one major drawback of this approach is that if the hypervisor or the guest uses superpages, the TLB would not benefit from them if the pages differ in size, i.e., there would be less TLB coverage than expected. This would result in more TLB misses and page-table walks and, naturally, a significant impact on the overall performance. To deal with the increased TLB coverage and TLB miss penalty caused by G-stage overhead, as well as with the inefficiency arising from the mismatch between translation sizes, we have augmented the CVA6 MMU with a large set-associative private unified L2 TLB as illustrated in Figure 21.

L2 TLB microarchitecture. The TLB stores each translation size in different structures to simplify the look-up logic. The L2 TLB follows a set-associative design with translation tags and data stored in SRAMs. As implemented in the GTLB, the replacement policy is also PLRU. To speed up the search logic, the L2

TLB look-ups and the PTW execute in parallel, thus not affecting the worst-case L1 TLB miss penalty and optimizing the L2 miss penalty. Moreover, the L2 TLB performs searches in parallel for each page size (4KiB or 2MIB), i.e., each page size translation is stored on different hardware structures with independent control and storage units. SFENCE and HFENCE instructions are supported, but a flush signal will flush all TLB entries, i.e., no filtering by VMIDs or ASIDs. We implemented the L2 TLB controller using a 4 state FSM. First, in *Flush* state, the logic encompasses walking through the entire TLB and invalidating all TLB entries. Next, in the *Idle* state, the FSM waits for a valid request or update signal from the PTW module. Third, in the *Read* state, the logic performs a read and tag comparison on the TLB set entries. If there is a hit during the look-up, it updates the correct translation and hit signals to the PTW. If there is no look-up hit, the FSM goes to the IDLE state and waits for a new request. Finally, in the *Update* state, we update the TLB upon a PTW update.

6.3.7 Sstc Extension

Timer registers are exposed as MMIO registers (*mtime*); however, the Sstc specification defines that *stimecmp and vstimecmp* are hart CSRs. Thus, we exposed the time value to each hart via connection to the CLINT. We also added the ability to enable and disable this extension at S/HS-mode and VS-mode via *menvcfg.STCE* and *henvcfg.STCE* bits, respectively. For example, when *menvcfg.stce* is 0, an S-mode access to stimecmp will trigger an illegal instruction. The same happens for VS-mode, when *henvcfg.stce* is 0 (throwing a virtual illegal instruction exception). The Sstc extension does not break legacy timer implementations, as software that does not support the Sstc extension can still use the standard SBI interface to set up a timer.

6.4 Design Space Exploration: Evaluation

In this section, we discuss the conducted DSE. The system under evaluation, the DSE configuration, the followed methodology, as well as benchmarks and metrics are described below. Each subsection then focuses on assessing the functional performance speedup per the configuration of each specific module: (i) L1 TLB (Section 6.4.1); (ii) GTLB (Section 6.4.2); (iii) L2 TLB (Section 6.4.3); and (iv) Sstc (Section 6.4.4).

System and Tools. We ran the experiments on a CVA6 single-core SoC featuring a 32KiB DCache and 16KiB ICache, targeting the Genesys2 FPGA at 100MHz. The software stack encompasses the OpenSBI (version 1.0), Bao hypervisor (version 0.1), and Linux (version 5.17). We compiled Bao using SiFive GCC version 10.1.0 for riscv64 baremetal targets and OpenSBI and Linux using GCC version 11.1.0 for riscv64 Linux targets. We used Vivado version 2020.2 to synthesize the design and assess the impact on FPGA hardware resources.

DSE Configurations. The DSE configurations are summarized in Table 19. For each module, we

selected the parameters we wanted to explore and their respective configurations. For instance, for the L1 TLB we fixed the number of entries as the design parameter and 16, 32, and 64 as possible configurations. Taking all modules, parameters, and configurations into consideration, it is possible to achieve up to 288 different combinations. Due to the existing limitations in terms of time and space, we carefully selected and evaluated 23 out of these 288 configurations.

Module	**Parameter**	**Configuration**		
		#1	#2	#3
L1 TLB	size entries	16	32	64
GTLB	size entries	8	16	–
L2 TLB	page size	4KiB	2MiB	4KiB+2MiB
	associativity	4	8	–
	size entries for 4KiB	128	256	–
	size entries for 2MiB	32	64	–
SSTC	status	enabled	disabled	–

Table 19: Design space exploration configurations.

Methodology. We focus the DSE evaluation on functional performance (execution cycles). We collected 1000 samples and computed the average of 95th percentile values to remove any outliers caused by some sporadic Linux interference. We normalized the results to the reference baseline execution (higher values translate to higher performance speedup) and added the absolute execution time on top of the baseline bar. Our baseline hardware implementation corresponds to a system with the CVA6 with hardware virtualization support but without the proposed microarchitectural extensions (e.g., GTLB, L2 TLB). We have also collected the post-synthesis hardware utilization; however, we omit the results due to lack of space (although we may occasionally refer to them during the discussion of the results in this section).

Benchmarks. We used the MiBench Embedded Benchmark Suite. The MiBench is an embedded application benchmark widely used in mixed-criticality systems to assess the performance [56, 131]. The MiBench incorporates a set of 35 application benchmarks grouped into six categories, targeting different embedded market segments. We focus our evaluation on the automotive subset. The automotive suite encompasses 3 highly memory-intensive benchmarks (*qsort*, *susan corners* and *susan edges*) that exercise many components across the memory hierarchy (e.g., MMU, cache, and memory controller). To complement our evaluation, we ran San Diego Vision Benchmark [581] in the selected designs for the PPA analysis. San Diego Vision is a computer vision benchmark widely used to evaluate embedded and mixed-criticality systems [16, 582]. It runs a full range of vision applications (e.g., motion tracking (*tracking*)) using multiple datasets with different sizes. Instructions to run our experiments with MiBench and San Diego Vision can be found here[5].

[5]https://github.com/ninolomata/bao-cva6-guide/tree/cva6-evaluation

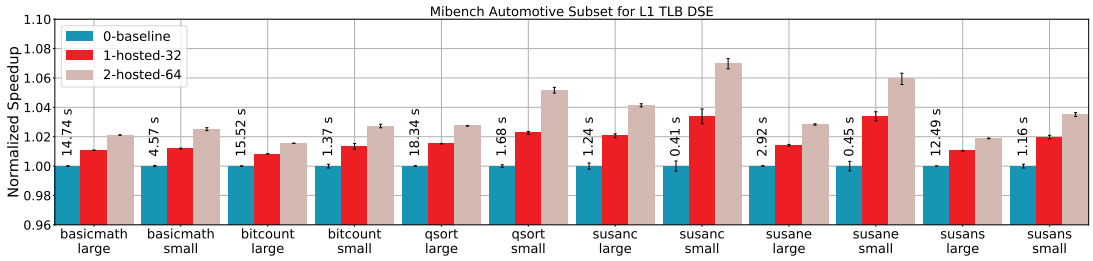


Figure 23: MiBench results for L1 TLB design space exploration evaluation.

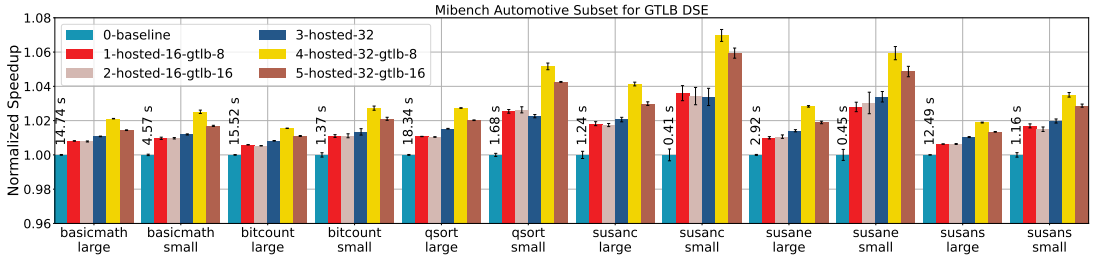


Figure 24: MiBench results for GTLB design space exploration evaluation.

6.4.1 L1 TLB

In this subsection, we evaluate the functional performance for a different number of L1 TLB entries, i.e., 16, 32, and 64.

L1 TLB Setup. To assess the L1 TLB functional performance speedup, we ran the full set of benchmarks for three different setups: (i) Linux virtual/hosted execution for the baseline *cva6-16* (*baseline*); (ii) Linux virtual/hosted execution for the *cva6-32* (*hosted-32*); and (iii) Linux virtual/hosted execution for the *cva6-64* (*hosted-64*).

L1 TLB Performance Speedup. Figure 23 shows the assessed results. All results are normalized to the *baseline* execution. Several conclusions can be drawn. Firstly, as expected, the *hosted-64* is the best-case scenario with an average performance increase of 3.6%. We can observe a maximum speedup of 7% in the *susanc* (small) benchmark and a minimum speedup of 2% in the *bitcount* (large) benchmark. However, although not explicitly shown in this paper, to achieve these results there is an associated impact of a 50% increase in the FPGA resources. We expected these results because the L1 TLB is a fully-associative TLB implemented as a full combinational circuit, and the CVA6 design is not optimized for this particular FPGA architecture. Secondly, the *hosted-32* configuration increases the performance by a minimum of 1% (e.g., basicmath large) and a maximum of 4% (e.g., *susanc* small), at a cost of about 15%-17% increase in area (FPGA). Finally, we can conclude that increasing the CVA6 L1 TLB size to 32 entries presents the most reasonable trade-off between functional performance speedup and hardware cost.

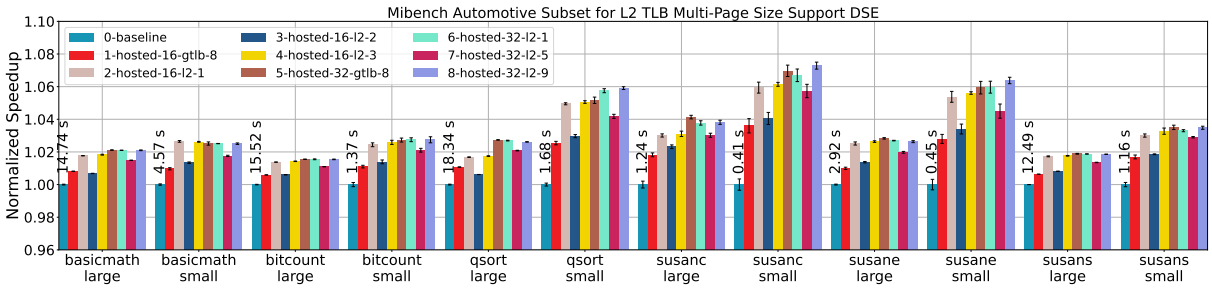


Figure 25: MiBench results for L2 TLB multi-page size support design space exploration evaluation.

6.4.2 GTLB

In this subsection, we assess the GTLB impact on functional performance. We evaluate this for a different number of GTLB entries (8 and 16) and L1 TLB entries (16 and 32).

GTLB Setup. To assess the GTLB functional performance speedup, we ran the full set of benchmarks for six distinct setups: (i) Linux virtual/hosted execution for the baseline *cva6-16* (*baseline*); (ii) Linux virtual/hosted execution for the *cva6-32* (*hosted-32*); (iii) Linux virtual/hosted execution for the *cva6-16-gtlb-8* (*hosted-16-gtlb-8*); (iv) Linux virtual/hosted execution for the *cva6-32-gtlb-8* (*hosted-32-gtlb-8*); (v) Linux virtual/hosted execution for the *cva6-16-gtlb-16* (*hosted-16-gtlb-16*); and (vi) Linux virtual/hosted execution for the *cva6-32-gtlb-16* (*hosted-32-gtlb-16*).

GTLB Performance Speedup. MiBench results are depicted in Figure 24. We normalized results to *baseline* execution. We highlight a set of takeaways. Firstly, the *hosted-16-gtlb-8* and *hosted-16-gtlb-16* scenarios have similar results across all benchmarks with an average percentage performance speedup of about 2%; therefore, there is no significant improvement from configuring the GTLB with 16 entries over 8 when the L1 TLB has 16 entries. Secondly, the *hosted-32-gtlb-8* setup presents the best performance, i.e., a maximum performance increase of about 7% for the *susanc* (small) benchmark; however, at a non-negligible overall cost of 20% in the hardware resources. Surprisingly, the hosted execution with 32 entries and a GTLB with 16 entries (*hosted-32-gtlb-16*) performs slightly worst compared with an 8-entry GTLB (*hosted-32-gtlb-8*). We have also collected microarchitectural hardware events on the CVA6. We noticed a slight increase in data cache misses for the *hosted-32-gtlb-16* compared to the *hosted-32-gtlb-8* of roughly 10% in some benchmarks (e.g., *susanc-large*), which explains why the GTLB with 16 entries performs worse than the GTLB with 8 entries. For instance, for the *susane* (large) benchmark the *hosted-16-gtlb-8* setup achieves a 6% performance increase while the *hosted-16-gtlb-16* only 5%. Finally, the *hosted-32* achieves a performance increase in line with *hosted-16-gtlb8* and *hosted-16-gtlb16* configurations.

6.4.3 L2 TLB

In this subsection, we assess the L2 TLB impact on functional performance. We conduct several experiments focusing on two main L2 TLB configuration parameters: (i) multi-page size support (4KiB or 2MiB or both) and (ii) TLB associativity. Furthermore, we also evaluated the L2 TLB impact in combination

Configuration	L1 TLB	GTLB	L2 TLB
cva6-16	16 entries	—	—
cva6-16-gtlb8	16 entries	8 entries	—
cva6-16-l2-1	16 entries	8 entries	4KiB - 128, 4-ways
cva6-16-l2-2	16 entries	8 entries	2MiB - 32, 4-ways
cva6-16-l2-3	16 entries	8 entries	4KiB - 128, 4-ways 2MiB - 32, 4-ways
cva6-32-gtlb8	16 entries	8 entries	—
cva6-32-l2-1	32 entries	8 entries	4KiB - 128, 4-ways
cva6-32-l2-2	32 entries	8 entries	4KiB - 128, 8-ways
cva6-32-l2-3	32 entries	8 entries	4KiB - 256, 4-ways
cva6-32-l2-4	32 entries	8 entries	4KiB - 256, 8-ways
cva6-32-l2-5	32 entries	8 entries	2MiB - 32, 4-ways
cva6-32-l2-6	32 entries	8 entries	2MiB - 32, 8-ways
cva6-32-l2-7	32 entries	8 entries	2MiB - 64, 4-ways
cva6-32-l2-8	32 entries	8 entries	2MiB - 64, 8-ways
cva6-32-l2-9	32 entries	8 entries	4KiB - 128, 4-ways 2MiB - 32, 4-ways
cva6-32-l2-10	32 entries	8 entries	4KiB - 256, 4-ways 2MiB - 64, 4-ways

Table 20: L2 TLB design space exploration configurations.

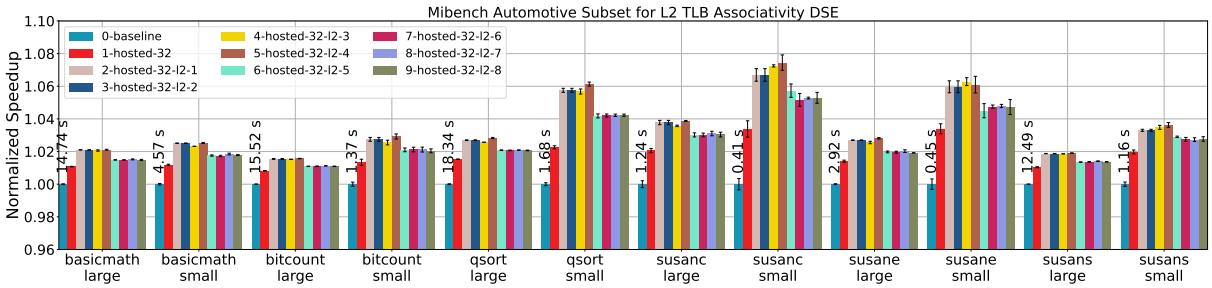


Figure 26: MiBench results for L2 TLB associativity design space exploration evaluation.

with different L1 TLB entries (16 and 32) and the GTLB with 8 entries. Table 20 summarizes the design configurations.

Multi-page Size Support Setup. To assess the performance speedup for the L2 TLB multi-page size, we have carried out a set of experiments to measure the impact of: (i) 4KiB page size (configurations *cva6-16-l2-1*,*cva6-32-l2-1*); (ii) 2MiB page size (configurations *cva6-16-l2-2*, *cva6-32-l2-5*); (iii) both 4KiB and 2MiB page sizes (configurations *cva6-16-l2-3*, *cva6-32-l2-9*); and (iv) increase the L1 TLB capacity. For each configuration, we run the MiBench benchmarks for ten different scenarios: (i) Linux virtual/hosted execution for the baseline *cva6-l1-16* (*baseline*); (ii) Linux native execution for the *cva6-16* (*bare*); (iii) Linux virtual/hosted execution for the *cva6-16-gtlb8* (*hosted-16-gtlb8*); (iv) Linux virtual/hosted execution for the *cva6-16-l2-1* (*hosted-16-l2-1*); (v) Linux virtual/hosted execution for the *cva6-16-l2-2* (*hosted-16-l2-2*); (vi) Linux virtual/hosted execution for the *cva6-16-l2-3* (*hosted-16-l2-3*); (vii) Linux virtual/hosted execution for

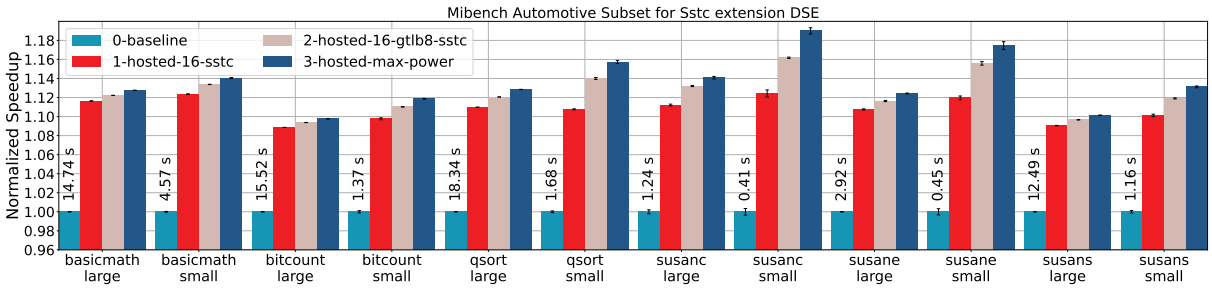


Figure 27: MiBench results for Sstc extension design space exploration evaluation.

the *cva6-32-gtlb8* (*hosted-32-gtlb8*); (viii) Linux virtual/hosted execution for the *cva6-32-l2-1* (*hosted-32-l2-1*); (ix) Linux virtual/hosted execution for the *cva6-32-l2-5* (*hosted-32-l2-5*); and (ix) Linux virtual/hosted execution for the *cva6-32-l2-9* (*hosted-32-l2-9*).

Multi-page Size Support Performance. MiBench results are depicted in Figure 25. All results were normalized to *baseline* execution. Based on Figure 25, we can extract several conclusions. First, configurations that support only 2MiB page size, i.e., *hosted-16-l2-2* and *hosted-32-l2-5*, present little or almost no performance improvement compared with the hardware configurations including only the GTLB (i.e., *hosted-16-gtlb-8* and *hosted-32-gtlb-8*). Second, supporting 4KiB page sizes causes a noticeable performance speedup for the L1 TLB with 16 entries, especially in memory-intensive benchmarks, such as *qsort* (small), *susanc* (small) and *susane* (small). The main reason that justifies this improvement lies in the fact that Bao is configured to use 2MiB superpages; however, the Linux VM uses mostly 4KiB page size, i.e., most of the translations are 4KiB. From a different perspective, we observe similar results for the *hosted-16-l2-1* and *hosted-32-gtlb-8*, with few exceptions in some benchmarks (e.g., *susane* (large), *susanc* (large) and *qsort* (large)). Moreover, for an L1 TLB configured with 32 entries, the average performance increase is roughly equal in less memory-intensive benchmarks (e.g. *basicmath* and *bitcount*). This is explained by the reduced number of L1 misses, due to its larger capacity which leads to fewer requests to the L2 TLB. Finally, we observe minimal speedup improvements on several benchmarks (e.g., *susans*) when supporting both 4KiB and 2MiB (*hosted-16-l2-3* and *hosted-32-l2-9*).

TLB associativity Setup. To evaluate the performance speedup for the L2 TLB associativity, we select eight designs from previous experiments and modified the following parameters: (i) the number of 4KiB entries (128 or 256) and 2MiB (32 or 64) entries for the L2 TLB; and (ii) the L2 TLB associativity, by configuring the L2 TLB in the 4-way or 8-way scheme. For each configuration, we have run the selected benchmarks for ten configurations: (i) Linux virtual/hosted execution for the baseline *cva6-l1-16* (*baseline*); (ii) Linux virtual/hosted execution for the *cva6-32* (*hosted-32*); (iii) Linux virtual/hosted execution for the *cva6-32-l2-1* (*hosted-32-l2-1*); (iv) Linux virtual/hosted execution for the *cva6-32-l2-2* (*hosted-32-l2-2*); (v) Linux virtual/hosted execution for the *cva6-32-l2-3* (*hosted-32-l2-3*); (vi) Linux virtual/hosted execution for the *cva6-32-l2-4* (*hosted-32-l2-4*); (vii) Linux virtual/hosted execution for the *cva6-32-l2-5* (*hosted-32-l2-5*); (viii) Linux virtual/hosted execution for the *cva6-32-l2-6* (*hosted-32-l2-6*); (ix) Linux virtual/hosted execution for the *cva6-32-l2-7* (*hosted-32-l2-7*); and (x) Linux virtual/hosted execution for the *cva6-32-l2-8*

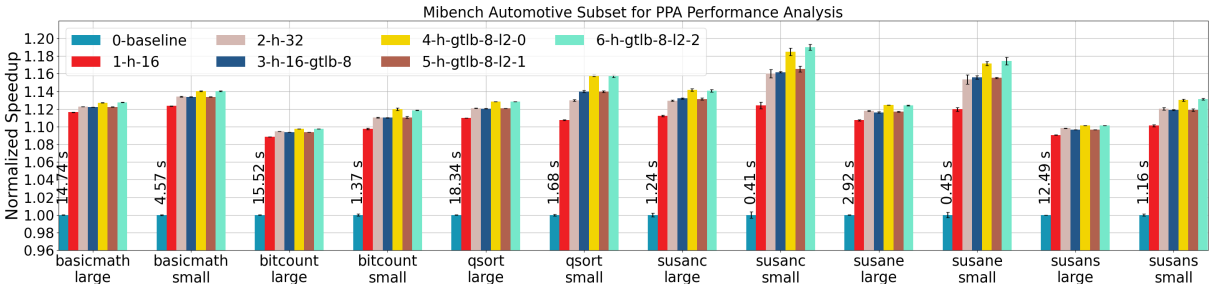


Figure 28: Summary of MiBench functional performance results for selected configurations

(*hosted-32-l2-8*).

TLB associativity Performance. The results in Figure 26 demonstrate that there is no significant improvement in modifying the number of sets and ways in 2MiB page size configurations (i.e., *hosted-32-l2-5* to *hosted-32-l2-8*). Additionally, we found that: (i) increasing the associativity from 4 to 8 or doubling the capacity in low memory-intensive benchmarks (e.g., *bitcount*) had little impact on functional performance; and (ii) memory-intensive benchmarks (e.g., *susanc* (small)) normally present a speedup when increasing the number of entries for the same associativity (although with a larger standard deviation). For instance, in the *susanc* (small) benchmark, we observe a performance speedup from 6% to 7% when the page support was 4KiB with 128- and 256-entry capacity organized into a 4-way set-associative (*hosted-32-l2-1* and *hosted-32-l2-3*).

6.4.4 Sstc Extension

Configuration	L1 TLB	GTLB	L2 TLB	SSTC
cva6-sstc	16	—	—	enabled
cva6-sstc-gtlb8	16	8	—	enabled
cva6-max-power	16	8	4KiB - 128, 4-ways 2MiB - 32, 4-ways	enabled

Table 21: SSTC design space exploration configurations.

In this subsection, we assess the Sstc extension impact on performance. We assess the Sstc performance speedup for a few configurations exercised in the former subsections. Table 21 summarizes the design configurations.

Sstc Extension Setup. To assess the Sstc extension performance speedup, we ran the full set of benchmarks for four different setups: (i) Linux virtual/hosted execution for the baseline *cva6-16* (*baseline*); (ii) Linux virtual/hosted execution for *cva6-sstc* (*hosted-16-sstc*); (iii) Linux virtual/hosted execution for *cva6-gtlb8-sstc* (*hosted-16-gtlb8-sstc*); and (iv) Linux virtual/hosted execution for *cva6-max-power* (*hosted-max-power-sstc*).

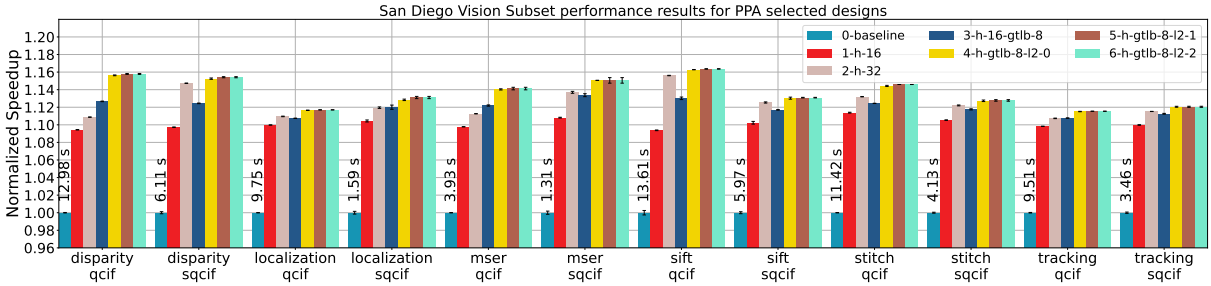


Figure 29: San Diego Vision Benchmark functional performance results for PPA analysis selected configurations

Sstc Extension Performance. The results depicted in Figure 27 show that for the *hosted-16-sstc* scenario, there is a significant performance speedup. For example, in the memory-intensive *qsort* benchmark, the performance speedup is 10.5%. Timer virtualization is a major cause of pressure on the MMU subsystem due to the frequent transitions between the Hypervisor and Guest. The "hosted-max-power" scenario performs the best, with an average performance increase of 12.6%, ranging from a minimum of 10% in the *bitcount* benchmark to a maximum of 19% in memory-intensive benchmarks such as the *susanc*. Finally, in less memory-intensive benchmarks, e.g., *basicmath* (large) and *bitcount* (large), the *hosted-16-gtlb8-sstc* scenario performs similarly to the *hosted-max-power* scenario, but with significantly less impact on hardware resources. For example, in the *basicmath* (large) benchmark, there is a negligible difference of 0.5 % comparing the *hosted-16-gtlb8-sstc* with the *hosted-max-power* scenario.

6.4.5 Designs for PPA Analysis Selection

We selected six configurations for the PPA analysis (see Table 22) based on the functional performance setup results summarized in Figure 28. We also ran San Diego Vision Benchmark to evaluate the selected designs with different workloads.

San Diego Vision Workloads Results. We ran the San Diego Vision Benchmark for the configurations elected for the PPA analysis. The results depicted in Figure 29 are in line with the results collected for the MiBench benchmark (Figure 28). First, for the *h-16* scenario, there is an average performance speedup of 10%. Second, the *h-32* outperforms *h-16-gtlb-8* in some benchmarks with few exceptions. For instance, for the *disparity qcif*, there is a slight difference of 2%. Third, *h-16-gtlb-8* has reasonable results with an average performance increase of about 12% (maximum of 14% and minimum of 10%). Finally, for the *h-gtlb-8-l2-0*, *h-gtlb-8-l2-1*, and *h-gtlb-8-l2-2* configurations, the performance is identical, with a pattern ranging a maximum of 16% and a minimum of 11%.

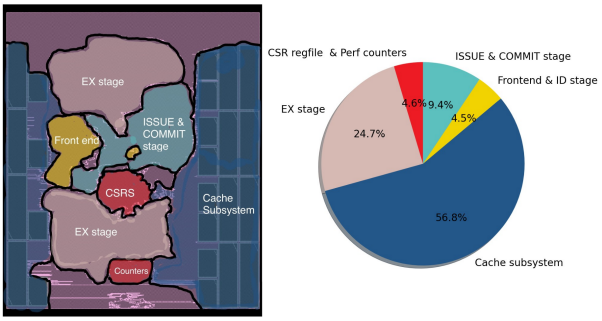


Figure 30: Baseline CVA6 layout 0.75mmx0.65mm

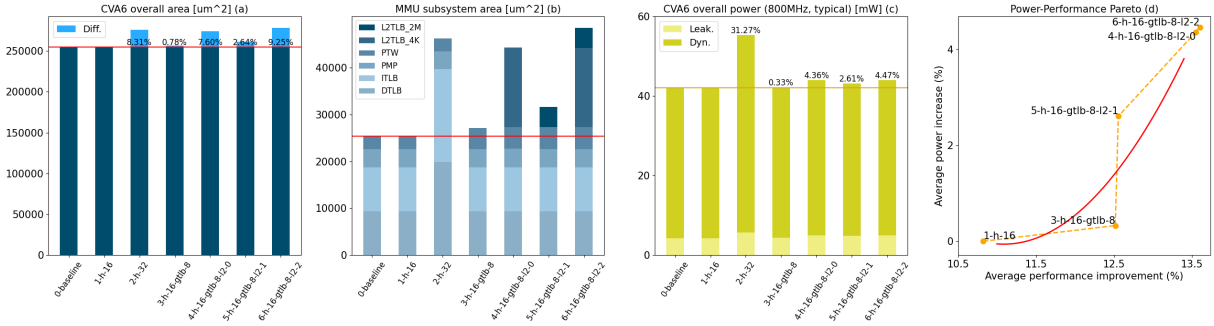


Figure 31: Area and power results

6.5 Physical Implementation

6.5.1 Methodology

Based on the functional performance results discussed in Section 6.4 and summarized in Figure 28, we select six configurations to compare with the baseline implementation of CVA6 with hardware virtualization support. Table 22 lists the hardware configurations under study. We implemented these configurations in 22 nm FDX technology from Global Foundries down to ready-for-silicon layout to reliably estimate their operating frequency, power, and area.

To support the physical implementation and the PPA analysis, we used the following tools: (i) Synopsys Design Compiler 2019.03 to perform the physical synthesis; (ii) Cadence Innovus 2020.12 for the place & route; (iii) Synopsys PrimeTime 2020.09 for the power analysis - extracting value change dump (VCD) traces on the post-layout; and (iv) Siemens QuestaSim 10.7b to simulate the parasitic-annotated netlist. Figure 30 shows the layout of CVA6, featuring an area smaller than 0.5mm^2.

6.5.2 Results

We fix the target frequency to 800MHz in the worst corner (SSG corner at 0.72 V, -40/125 °C). All configurations manage to reach the target frequency. We then compare the area and power consumption while running a dense 16x16 FP matrix multiplication at 800MHz with warmed-up caches (TT corner, 25 °C). Leveraging the extracted power measurements and the functional performance on the MiBench

	SSTC support	I/DTLB #entries	8-entries GTLB	4KiB L2 TLB	2MiB L2 TLB
0-vanilla	x	16	x	x	x
1-h-16	✓	16	x	x	x
2-h-32	✓	32	x	x	x
3-h-gtlb-8	✓	16	✓	x	x
4-h-gtlb-8-l2-0	✓	16	✓	✓	x
5-h-gtlb-8-l2-1	✓	16	✓	x	✓
6-h-gtlb-8-l2-2	✓	16	✓	✓	✓

Table 22: 7 selected configurations for PPA analysis in GF22nm.

benchmarks, we further obtain the relative energy efficiency.

Figure 31 depicts the PPA results. Figure 31(a) shows that the Sstc extension has negligible impact on the area. In fact, as expected, the MMU is the microarchitectural component with a higher impact on power and area. Figure 31(b) highlights the MMU area comparison. The configuration with 32 entries doubles the ITLB and DTLB area, while the other configurations have little to no impact on the ITLB and DTLB; they, at most, add the GTLB and the L2-TLB modules on top of the existing MMU configuration. Figure 31(c) shows the measured power consumption. We observe that increasing the number of L1 TLB entries from 16 to 32 (2-h-32) increases the power by 31%, while the other configurations increase power by less than 5% compared with the vanilla CVA6. Figure 32 shows the relative energy efficiency on the MiBench benchmarks. The second configuration (2-h-32) is less energy efficient than the baseline since the performance gain ($\leq 15\%$) is smaller than the power increase ($\geq 30\%$). It is therefore excluded in the graph analysis. On the other hand, all the other configurations increase energy efficiency by up to 16%.

Figure 31(d) plots the measured power consumption of the energy-efficient configurations against the average performance improvement on the MiBench benchmarks. The SSTC extension alone brings an average performance improvement of around 10% with a negligible power increase. At the same time, the explored MMU configurations can offer a clean trade-off between performance and power. The most expensive configuration can provide an additional 4% performance gain for a 4.47% power increase. Lastly, the hardware configuration including the Sstc support and a GTLB with 8 entries (3-h-16-gtlb-8), is the most energy-efficient one, with the highest ratio between performance and power increase.

In conclusion, we can argue that the hardware configuration with Sstc support and a GTLB with 8 entries (3-h-16-gtlb-8) is the optimal design point since we can achieve a functional performance speedup of up to 16% (approx. 12.5% on average) at the cost of 0.78% in area and 0.33% in power.

6.6 Discussion

Performance Evaluation. We carried out an extensive functional performance evaluation for multiple FPGA design configurations in a single-core environment. As part of future work, we plan and encourage

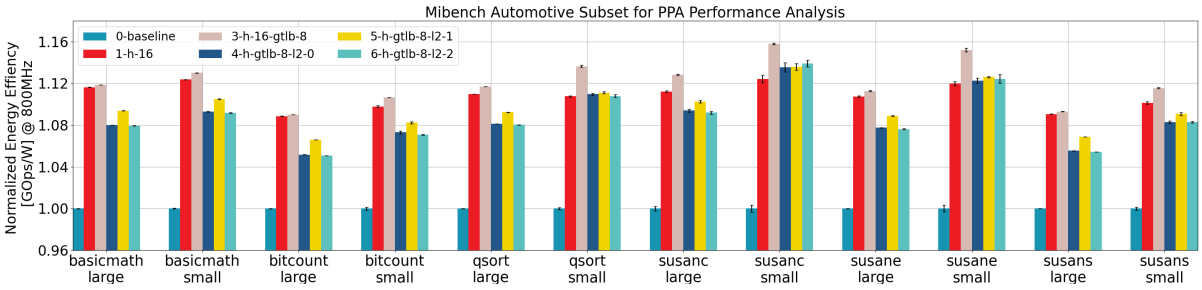


Figure 32: Energy efficiency results.

the evaluation of additional configurations and software stacks. Possible directions include (i) experimenting with other hypervisors like XVisor [571] in different configurations (e.g., having two guests running on a single-core); (ii) running other commercial benchmarks like SPEC17; and (iii) shifting to multicore environments, e.g., available in frameworks such as OpenPiton [583].

System-level Virtualization. The RISC-V ISA has reached a new state of maturity, and some of its previously existing gaps [56] (e.g., no standard interrupt controller with interrupt virtualization support) are already addressed. The AIA [580] specification has been ratified, and it provides preliminary support for interrupt virtualization using MSI. Hypervisors can leverage the AIA to reduce interrupt latency to guests and improve overall performance. As part of our ongoing efforts, we are working on providing an open-source reference implementation of the AIA IPs. The RISC-V IOMMU [6] is also now ratified. Fundamentally, the IOMMU protects memory accesses from DMA-capable devices while guaranteeing isolation between VMs and the hypervisor. For instance, in Bao, all devices are assigned in a one-to-one scheme to a single guest using pass-through device mechanisms, which means Bao entirely relies on the IOMMU to do this. As part of our ongoing efforts, we are working on providing open-source reference implementations of the AIA IPs and the IOMMU IP.

Memory Subsystems Improvements. From an MMU microarchitectural point of view, there is ample room for improvement. We demonstrated that the CVA6 memory subsystem is the major cause of performance degradation, and further improvements are of utmost importance. For instance, we could redesign the CVA6 MMU to support TLB coalescing by augmenting the PTE to support NAPOT pages (Snapot extension). Although we have accelerated the PTW with a dedicated TLB for the second stage, a few memory accesses to the memory hierarchy are still needed, i.e., side-effects are still expected on caches and interconnect memory bandwidth. One could explore using a PTE cache to store PTE entries close to the PTW and thus reduce the number of memory accesses and cache pollution. From the platform standpoint, RISC-V's rich ecosystem offers extensions that can benefit virtualized environments. We already discussed the standard cache management instructions, i.e., CMOs, defined in the Zicbom and Zicboz extensions. Hypervisors like Bao depend upon such instructions to implement partitioning mechanisms such as cache coloring, which otherwise would have to flush caches using firmware *ecalls*. Finally, this work targeted a single-core platform with no LLC. Hence, the effectiveness of such mechanisms is yet

[6] https://github.com/riscv-non-isa/riscv-iommu

to be proved in a multicore environment where inter-core interference is visible throughout the memory hierarchy. An all-new class of security challenges is present, e.g., timing side-channels [556].

6.7 Related Work

The RISC-V hypervisor extension is a relatively new addition to the privileged architecture of the RISC-V ISA, ratified as of December 2021. Notwithstanding, from a hardware perspective, there are already a few open-source and commercial RISC-V cores with hardware virtualization support (or ongoing support): (i) the open-source Rocket core [584]; (ii) the space-grade NOEL-V [576, 585], from Cobham Gaisler; (iii) the commercial SiFive P270, P500, and P600 series; (iv) the commercial Ventana Veryon V1; (v) the commercial Dubhe from StarFive; (vi) the open-source Chromite from InCore Semiconductors; and (vii) the open-source Shakti core from IIT Madras.

Processor	Pipeline	Priv. Hyp.	SSTC	2D-MMU	L1 Cache	L1 TLB	L2 TLB	PTW Opts	Status
Rocket	5-stage in-order	v0.6	X	SV39x4	32 KiB, I/DCache	set-assoc. 4KiB I/DTLB or full-assoc. superpages I/DTLB, 4-entries (configurable)	dir.-mapped, (configurable size)	PTE Cache	Done
NOEL-V	7-stage dual-issue in-order	v0.6	✓	SV32x2 SV39x4	32 KiB I/DCache	I/DTLB, (configurable)	X	hTLB	Done
Chromite	6-stage in-order	v1.0	?	SV32x4 SV39x4 SV48x4 SV57x4	32 KiB I/DCache	set-assoc. split I/DTLB, (configurable page sizes support) or full-assoc. I/DTLB, ?-entries (def. 4)	X	X	WIP
Shaktii C Class	5-stage in-order	v1.0	?	SV32x4 SV39x4 SV48x4	16 KiB I/DCache	fully assoc. L1 I/DTLB, 4-entries	X	X	WIP
CVA6	**6-stage in-order**	**v1.0**	✓	**SV39x4**	**32 KiB DCache, 16KiB ICache**	**full-assoc. I/DTLB, 4-entries**	**set assoc. 4KiB 128-256 entries, 4-8 ways or/and set assoc. 2MiB 32-64 entries, 4-8 ways**	**GTLB**	**Done**

Table 23: CVA6 alike RISC-V Cores with Hypervisor Extension support features: ✓- supported; X - not supported; and ? - no information available.

Table 23 summarizes some information about the open-source cores, focusing on their microarchitectural and architectural features relevant to virtualization. Table 23 shows that RISC-V cores with full support for the hypervisor extension (e.g., the Rocket core and NOEL-V) are only compliant with version 0.6 of the specification. The work described in this paper is already fully compliant with ratified version 1.0. Regarding the microarchitecture: (i) the Rocket core has some general MMU optimizations (e.g., PTE cache and L2 TLB), none of which are specifically targeting virtualization (like our GTLB); (ii) Chromite and Shaktii have no MMU optimizations for virtualization (and their support for virtualization is still in progress); and (iii) NOEL-V has a dedicated hypervisor TLB (hTLB). From a Cache hierarchy point of view, all cores have identical sizes of L1 Cache, except the CVA6 ICache, which is smaller (16KiB).

In terms of functional performance, our prior work on the Rocket Core [56] concluded an average of 2% performance overhead due to the hardware virtualization support. Findings in this work are in line with the ones presented in [56] (however, with a higher penalty in performance due to the area-energy focus of the microarchitecture of the CVA6). Notwithstanding, we were able to optimize and extend the

microarchitecture to significantly improve performance at a fraction of area and power. Results in terms of performance are not available for other cores, i.e., NOEL-C, Chromite, and Shaktii. From an energy and area breakdown perspective, none of these cores have made a complete public PPA evaluation. NOEL-V only reported a 5% area increase for adding the hypervisor extension.

A large body of literature also covers techniques to optimize the memory subsystem. Some focus on optimizing the TLB [586–594], while others aim at optimizing the PTW [595, 596]. For instance, in [596], the authors proposed a dedicated TLB on the PTW to skip the second translation stage and reduce the number of walk iterations. Our proposal for the GTLB structure is similar. Nevertheless, to the best of our knowledge, this is the first work to present a design space exploration evaluation and PPA analysis for MMU microarchitecture enhancements in the context of virtualization in RISC-V, fully supported by a publicly accessible open-source design [7].

6.8 Conclusion

This article reports our work on hardware virtualization support in the RISC-V CVA6 core. We start by describing the baseline extension to the vanilla CVA6 to support virtualization and then focus on the design of a set of enhancements to the nested MMU. We first designed a dedicated G-Stage TLB to speed up the nested-PTW on a TLB Miss. Then, we proposed an L2 TLB with multi-page size support (4KiB and 2MiB) and configurable size. We carried out a design space exploration evaluation on an FPGA platform to assess the impact on functional performance (execution cycles) and hardware. Then, based on the DSE evaluation, we elected a few hardware configurations and performed a PPA analysis based on 22nm FDX technology. Our analysis demonstrated several design points on the performance, power, and area trade-off. We were able to achieve a performance speedup of up to 19% but at the cost of a 10% area increase. We selected an optimal design configuration where we observed an average performance speedup of 12.5% at a fraction of the area and power, i.e., 0.78% and 0.33%, respectively.

[7] https://github.com/minho-pulp/cva6/tree/feature-hyp-ext-final

7

CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation

This chapter was submitted as:

Bruno Sá, Donato Ferraro, Everton de Matos, Andrea Bastoni and Sandro Pinto. "*CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation*", Submitted to *IEEE Transactions on Computers*.

Abstract

CHERI technology has gained impressive momentum in recent years due to its inherent security features, which promote the development of safer and more trustworthy software and platforms. RISC-V has emerged as the ideal platform for advancing CHERI technology, mainly due to its open-standard framework and robust community support.

In this article, we present our work on integrating the CHERI capability architecture support into the RISC-V CVA6 core. Our contributions are diverse, encompassing microarchitecture design, platform-level implementation of CHERI capabilities, and comprehensive performance evaluations using real-world benchmarks. We successfully implemented the CHERI extension on top of the CVA6 baseline with hypervisor extension support and developed an open-source Tag Controller for managing capability tags. This is the first RISC-V CPU with hardware virtualization and CHERI extension support. We gathered functional performance results by executing the MiBench and San Diego Vision benchmarks in a single-core configuration. We observed a performance slowdown of approximately 20% on average when comparing the legacy code and pure-capability code running atop CheriBSD, accompanied by a resource increase of 50%. Lastly, we made our work publicly available as open source, inviting the community to explore alternative design configurations and software stacks.

7.1 Introduction

Memory safety vulnerabilities remain among the most persistent software vulnerabilities in applications written in memory-unsafe programming languages, such as C/C++. Studies conducted by Microsoft [161] and Chromium [403] reported that roughly 70% of all security bugs are related to memory safety issues. Despite the advances in safe languages (e.g., Rust programming language) and static analysis tools [190, 405], most critical software is written in C/C++ [190], requiring non-negligible engineering effort to port to a safe language [190, 408].

In addition, another notable concern is the increasing complexity and interconnectivity of modern applications, which often include a collection of untrusted components (third-party libraries, applications, services). To tackle this issue, several compartmentalization technologies [597, 598] have been proposed to isolate and break down complex software into restricted pieces that run in isolation, limiting the privileges it can obtain and preventing a compromised component from breaking the entire system. Traditional approaches, such as page-based protection, provide only coarse-grained protection for process or VM isolation but are impractical to scale down for more fine-grained protection and add substantial runtime overhead to achieve intra-process level isolation [598]. The lack of fine-grained isolation opens the attack surface, which means that a single component can compromise the entire system [8].

CHERI [37, 394] is a capability-based security protection model that addresses memory safety and compartmentalization through a unified hardware-software co-design model. CHERI extends commodity ISAs, such as RISC-V and Arm, with architectural capabilities that are unforgeable tokens of authority.

These capabilities provide fundamental primitives capable of enforcing fine-grained memory control and low-overhead software compartmentalization support.

In this work, we share our journey towards implementing CHERI's architectural and microarchitectural support in an open-source RISC-V CVA6-based [567] (64-bit) SoC. At the architectural level, the implementation is compliant with the CHERI ISA extension (version 9) from Cambridge, which is the baseline CHERI ISA used to define CHERI RISC-V standard extensions (`Zcheri` extensions [107]). At the microarchitectural level, we extended CVA6 to support the CHERI RISC-V ISA by adding capability metadata to the register file, capability-enforced memory and control-flow operations, tagged memory, and support for capability manipulation instructions. In addition, we integrated a `TagController` near memory to manage capability tag storage and ensure correct propagation throughout the memory hierarchy. We also present and discuss a comprehensive performance overhead evaluation using three widely known benchmark suites, CoreMark/MHz [599], MiBench [600], and San Diego Vision [581]. We deployed the design on a Genesys 2 FPGA platform to evaluate its functional performance in terms of execution cycles and to quantify the associated hardware resource overhead.

As of this writing, the CHERI RISC-V ISA is undergoing ratification [107]. To the best of our knowledge, although several CHERI capability-architecture CPU implementations exist in BlueSpec Verilog, no mid- or high-end open-source RISC-V cores with CHERI support are written in SystemVerilog. This limits the dissemination of the technology within the research community, as many widely used open-source RISC-V cores are developed in SystemVerilog. Furthermore, no existing open-source CHERI-capable RISC-V core provides simultaneously CHERI and virtualization support, which (i) are already available in Arm with Morello [167] and (ii) are key to ratifying the CHERI ISA extension in RISC-V. We selected the CVA6 core primarily due to its widespread adoption in both industry and academia, and we anticipate that our CHERI-enabled CVA6 implementation will contribute to a suite of reference designs for validating the standard CHERI RISC-V ISA extension.

To summarize, with this work, we make the following contributions. First, we present the first hardware implementation of CHERI integrated with the RISC-V hypervisor extension, enabling capability-based memory protection in virtualized environments. In particular, we implement CHERI support in the 64-bit version of the open-source CVA6 core and port the CHERI capability library from Bluespec Verilog to SystemVerilog (Section 7.3). Second, we propose and implement the `TagController` module, which manages the tagged memory metadata (Section 7.4). Third, we integrate the verification framework TestRIG into the CVA6 pipeline, ensuring alignment with state-of-the-art CHERI core verification practices (Section 7.3.10 and 7.5.1). Finally, we design and evaluate a CVA6-based SoC that combines CHERI, hypervisor support, and tagged memory (Section 7.5). Specifically, we conducted a comprehensive performance evaluation, using MiBench and San Diego Vision benchmarks and reported the hardware cost in terms of FPGA resources. All contributions are released as open source to support further research and enable reproducibility in the RISC-V security and virtualization-related groups[1].

[1] https://github.com/zero-day-labs/cheri-cva6

7.2 Background

This section covers the background related to capability-based protection, RISC-V technology (ISA, extensions, and cores) and CHERI ISA.

7.2.1 A Brief History of Capability-based Protection

In the mid-1960s, Dennis and Van Horn [66] introduced a capability-based protection model for multiprogrammed systems, where capabilities are `unforgeable tokens of authority` that confer fine-grained access to system objects (e.g., read, write, execute). In capability-based systems, an application's authority is defined by the set of capabilities it holds. Compared to ACLs [70], capabilities inherently enforce the `Principle of Least Privilege` [143] by granting access only to explicitly held rights, minimizing unnecessary authority. Capabilities are unforgeable, encode access rights directly, and eliminate the need for centralized enforcement, mitigating issues such as the confused deputy problem [71]. Numerous software and hardware implementations emerged through the mid-1980s [67, 68, 77, 152], with renewed interest in CHERI's [37] principled approach to building trustworthy systems.

7.2.2 RISC-V Privileged Specification

The RISC-V privileged ISA [177] defines three privilege levels: (i) `machine mode` (M-mode), typically used to run firmware implementing the SBI, such as OpenSBI; (ii) `supervisor mode` (S-mode), intended for general-purpose operating systems that require features like virtual memory; and (iii) `user-mode` (U-mode), where user-level applications execute. Each privilege level has access to a defined set of CSRs to manage aspects such as exception handling, interrupt delegation, and memory translation. The modularity of the RISC-V ISA enables implementations across a wide range of design points, from minimal embedded-class systems supporting only M-mode to full-featured systems supporting M/S/U modes. Additionally, RISC-V provides architectural support for virtualization via the RISC-V Hypervisor extension [56].

7.2.3 CHERI ISA in Nutshell

CHERI boosts traditional ISAs with capability primitives to enhance security and prevent future and known vulnerabilities [37]. CHERI defines architectural support for capabilities that enforce two fundamental primitives: (i) byte-level memory protection by replacing traditional integer pointers with capabilities as fat pointers that restrict memory access and code execution at the instruction level by encoding the memory bounds and permissions on the capability and (ii) fine-grained compartmentalization (function-level) using capability primitives to isolate intra-address space software components, such as the application libraries.

Architectural Capability. Figure 33 depicts the CHERI capability format for a 64-bit architecture. Capabilities are 128-bit entities, twice the size of a conventional pointer structure, that encode the address, bounds, permissions, and additional metadata in a compressed layout [81]. This compression leverages a

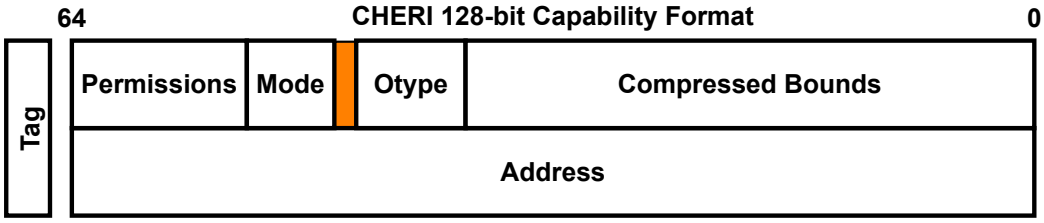


Figure 33: CVA6 CHERI capability 128-bit compressed format.

floating-point-like encoding scheme to represent the capability bounds, trading some spatial precision for a wider representable range. To ensure runtime integrity, each capability is associated with an out-of-band 1-bit tag, which unambiguously distinguishes valid capabilities from raw data and prevents unintended or malicious reuse of pointer-like values.

Capability Properties. CHERI architectural capabilities must uphold some security properties using guarded manipulation and a 1-bit tag per capability. These mechanisms prevent the forging of capabilities (`unforgeable`) by allowing new capabilities to be derived only from valid capabilities (`integrity and valid provenance`), subject to non-escalation of privileges (`monotonicity`). Furthermore, capabilities restrict memory access by constraining the pointer's bounds (`bounds checking`) and available permissions (`permissions checking`). Attempts to violate the properties above either cause an exception or invalidate the capability by clearing the tag.

Tagged Memory. During program execution, capabilities are transferred between registers and memory, with hardware transparently managing their associated tag bits to ensure integrity. These tags, stored out-of-band, propagate through the memory hierarchy and are maintained at each level [434]. To support this, systems either: (i) repurpose DRAM ECC bits to store tag metadata inline, efficient but not always feasible on low-power memory; or (ii) allocate a shadow bitmap in DRAM (one bit per capability-aligned word), managed by a tag controller near memory to intercept accesses and cache tags efficiently.

Hybrid Protection Model. CHERI is designed for compatibility with traditional isolation mechanisms (e.g., MMU) and supports incremental adoption without requiring complete software rewrites. It enhances intra-process isolation with fine-grained protection over virtual addresses. In hybrid systems, CHERI-aware and legacy code can coexist. This is achieved by: (i) constraining control flow via the PCC, (ii) limiting memory access from legacy pointers using the DDC, and (iii) enforcing protection over virtual addresses. CHERI supports two execution modes: pure-capability (all pointers are capabilities) and hybrid (selective pointer annotation).

Object Sealing Mechanism. The CHERI `Otype` field allows capabilities to be sealed and treated as immutable objects that cannot be dereferenced or modified. These mechanisms are the foundation for providing intra-address space software compartmentalization.

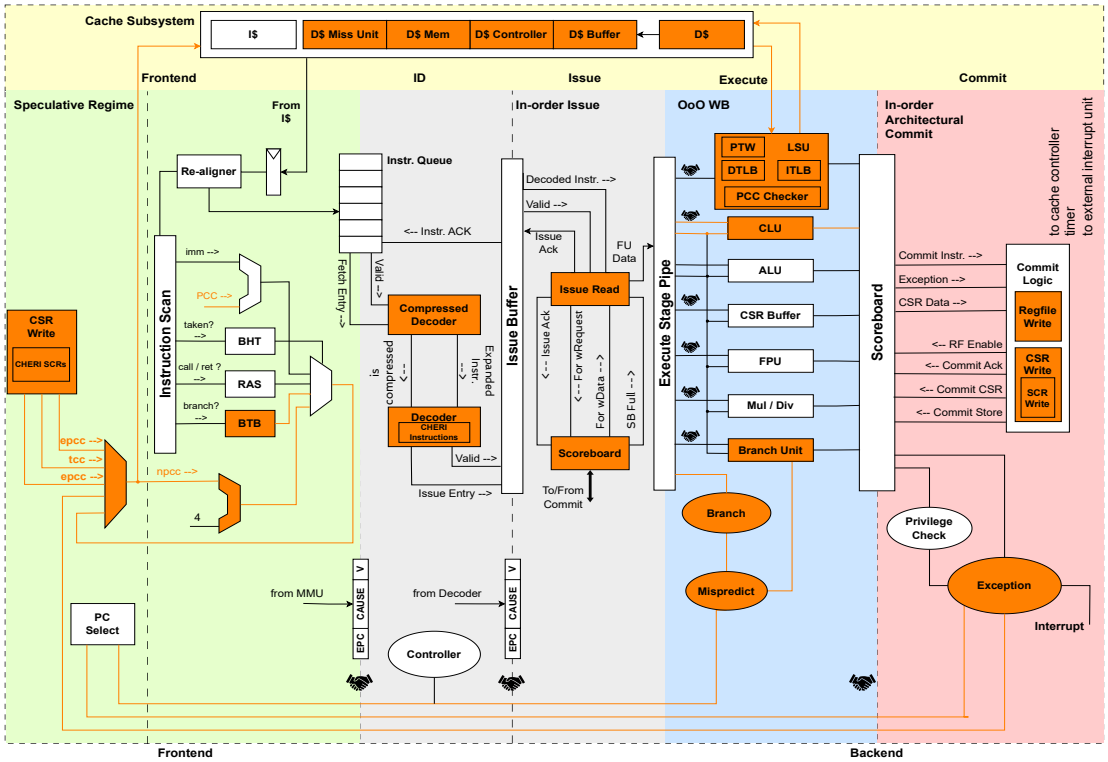


Figure 34: CVA6 pipeline microarchitecture featuring the RISC-V CHERI extension. Highlighted in orange are the major microarchitectural changes to the core functional blocks. Adapted from [567].

7.2.4 RISC-V CVA6 CPU

CVA6 [567] is a highly configurable, open-source RISC-V core designed for the application-class, that implements the RISC-V RV64 and RV32 ISAs. It supports the standard three privilege execution M/S/U along with hardware virtualization support (Hypervisor extension) [112], and hardware memory virtualization through the implementation of a Memory Management Unit (MMU). These features make the core well-suited for running fully-fledged operating systems like Linux as well as type-1 and type-2 hypervisors (e.g., Bao [118]). Additionally, the core offers various design configurations tailored to specific workloads, including a superscalar design. From a microarchitectural perspective, the CVA6 incorporates a 6-stage pipeline with options for single or dual instruction issue and an Out-of-Order (OoO) execution stage.

7.3 CVA6 "From Scratch" CHERI Support

This section presents the design modifications to the CVA6 core to implement the RISC-V CHERI version 9 (Cambridge) ISA [394]. To our knowledge, this is the first full hardware integration of CHERI into CVA6, making it the first open-source RISC-V core to simultaneously support CHERI and virtualization. Figure 34 presents the modified CVA6 pipeline. The following subsections detail the microarchitectural changes required to implement CHERI support in CVA6.

Name	Description
cap_reg_t	Capability definition in a register.
cap_mem_t	Capability definition in memory.
set_cap_reg_bounds	Set capability bounds in a register.
cap_reg_inc_offset	Increment capability offset.
set_cap_reg_addr	Set address (no representability check).
set_cap_reg_address	Set address (with representability check).
*get_cap_reg_**off/len***	Retrieve offset and length fields.
*get_cap_reg_**top/base***	Retrieve top, base, and metadata fields.
get_cap_reg_metadata	Retrieve decode bounds metadata fields e.g., correction bits (*cb* and *ct*).
is_cap_reg_inbounds	Check if capability is in bounds.
***decode/encode**_bounds*	Convert compressed and raw bounds.
cap_mem_to_cap_reg	Convert memory to register capability.
cap_reg_to_cap_mem	Convert register to memory capability.
seal/unseal	Seal or unseal a capability.

Table 24: CHERI Capability RTL library interface in SystemVerilog.

7.3.1 Capability Library Port to SystemVerilog

As part of our effort to enable CHERI on the CVA6 core, we ported the CHERI reference implementation of the CHERI CapLib [601] RTL API, initially written in Bluespec Verilog, to SystemVerilog ensuring compliance with the CHERI Concentrate 128 capability specification [394]. This API abstracts several complex, low-level capability management operations, such as setting capability bounds, into a standard interface used across multiple implementations. Table 24 lists the fundamental functions and data types that we provide in our CHERI capability RTL library implementation.

7.3.2 Capabilities in Pipeline

Our design defines three capability formats tailored to different pipeline stages and logic units: (i) capability in-memory format, (ii) register format, and (iii) full capability representation. Figure 35 illustrates each capability format used in the pipeline. In-memory capabilities follow the CHERI Concentrate compression scheme [81], yielding a 128-bit representation with an out-of-band tag bit to minimize cache and memory footprint. This format is used to store capabilities in the main memory and all memory-hierarchy subsystems (e.g., L1 Cache). Upon loading capabilities from memory, the capability is decompressed into register format bounds using the `CapLib` decompression function (Section 7.3.1). We use this format to store capabilities in the register files and all other pipeline structures, except o in the `Execute Stage`, where full metadata access is required for operations such as bounds checks and metadata inspection. Here, we expand the capability into its full representation, which includes additional correction bits to accurately recover bounds information [81].

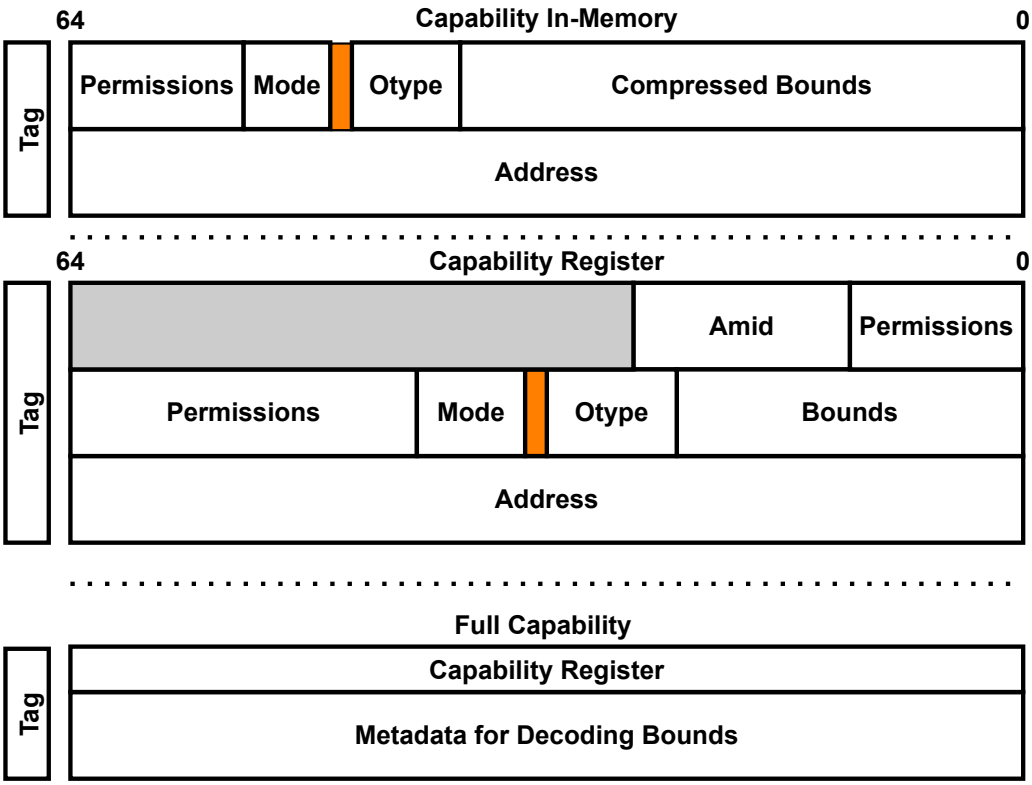


Figure 35: CVA6 CHERI capability in-pipeline formats.

7.3.3 CHERI CSRs, SCRs and CHERI Register File

The core logic for handling CHERI CSRs and Special Capability Registers (SCRs) resides in the `CSR Regfile` module. The SCR logic mirrors that of standard CSRs but redirects accesses to baseline registers (e.g., `mtvec`, `mepc`) to the address field of their corresponding CHERI SCRs. To interface with SCRs, we implemented the `CSpecialRW` instruction that enables explicit reads and writes to SCRs. Table 25 summarizes the current CHERI feature support in CVA6.

Capability Register File. We adopt a merged register file design that extends the baseline CVA6 register file to accommodate both integers and capabilities. This unified approach simplifies pipeline integration and avoids duplicating register logic, resulting in a more area-efficient implementation.

7.3.4 Hybrid Mode Support

The design supports both hybrid (integer) and pure-capability execution modes to maintain compatibility with non-CHERI applications. Execution mode is controlled via the `PCC.mode` bit: when cleared, the processor operates in integer mode, where (i) operands and results are standard integers, (ii) memory accesses are constrained by the `DDC`, (iii) code execution is still governed by the `PCC`, and (iv) a minimal

SCRs	*pcc*	●
	ddc	●
	xttc/xtdc/xscratchc/xepcc	●
CSRs	*xccsr*	●
	xtval	●
	bit 28 in *xenvcfg*	●
Intructions	capability-inspection	●
	capability-modification	●
	pointer-arithmetic	●
	pointer-comparison	●
	control-flow	●
	special capability access register	●
	fast-register clearing	◐
	tag-memory access	◐
	memory load/stores with explicit address type	●
	(atomic) memory-access via capability	●
Exceptions	Capability Exception	●
	Load/Store capability fault	●
MMU Extensions	Store capabilities barrier (PTE bits *CW* and *CD*)	●
	Load capabilities barrier and strip tag function (PTE bits *CR*, *CRM*, *CRG* bits)	●
Additional Features	*pcc* and *ddc* relocation	○
	capability mode flag	●

Table 25: CHERI extension features implemented in the CVA6 core: ● fully-implemented; ◐ partially implemented; ○ not implemented.

subset of capability instructions is available for operations involving explicit use of capabilities in integer mode or pure mode. In integer mode, software may explicitly invoke capability instructions, but legacy accesses are not interpreted as offsets relative to the `DDC` or `PCC`, as relocation of these capabilities is not supported.

7.3.5 CHERI Instructions

The CHERI-RISC-V ISA extends the base ISA with capability-aware instructions, grouped into four categories (see Table 25): (i) capability inspection and modification, (ii) control-flow operations (e.g., `CJAL`, `CJALR`),

(iii) memory access via capabilities, and (iv) miscellaneous instructions (e.g., tag inspection, pointer arithmetic, sealing). Supporting these required changes across several CVA6 functional units, specifically the `Cheri Logic Unit (CLU)`, `Decode Unit`, `Branch Unit`, and `Load/Store Unit`.

CHERI Logic Unit. The `CLU` is a single-cycle unit in the execution stage that implements the capability inspection, modification, and miscellaneous instructions. It handles operations such as bounds and permission checks, offset adjustment, setting the capability bounds, sealing/unsealing, and exception detection.

CHERI-aware Branch Unit. We extended the `Branch Unit` to support capability-based jumps. This unit updates and performs bounds, permissions, tag, and type checks on `PCC`, such as preventing jumps using a sealed capability. To ensure consistency with predicted control flow, it compares the predicted and actual `PCC` values to detect mispredictions. The Branch Predictor predicts the full `PCC` to achieve optimal performance. To ensure the correctness of the predicted control flow, the logic compares the predicted `PCC` with the computed next `PCC`, including the capability metadata and address.

CHERI-aware Load/Store Unit. The `Load/Store Unit` was adapted to support capability-based memory accesses, including atomic operations. Enhancements include: (i) bounds, permissions, tag, and seal checks, (ii) `DDC`-based restriction for legacy integer-mode accesses, and (iii) logic for encoding/decoding capabilities between in-memory capability and register formats, using XOR with a null capability to ensure consistency with the zeroed representation of C-language `NULL` pointers in memory, including the tag bit.

7.3.6 PCC Handling

The `PCC` extends the traditional Program Counter (PC) with bounds and permissions. In the baseline CVA6, the PC is speculatively predicted early in the pipeline by the frontend. To support CHERI, we extended the `Frontend` logic and `Branch Predictor` to operate on full `PCC` values rather than raw fetch addresses. Additionally, the MMU logic was enhanced to perform `PCC` bounds and permissions checks in parallel with address translation during instruction fetch.

7.3.7 CHERI MMU Extensions

We extended the CVA6 MMU to support CHERI-specific page-table enhancements and enforce capability-aware memory access constraints. The CHERI-RISC-V specification introduces five additional bits in the page table entry (see Table 25) to assist with capability tracking and revocation [391, 393]. The MMU interprets these bits and triggers exceptions as needed. For instance, if the Capability Write (`CW`) bit is cleared, capability stores are disallowed, and an exception is raised.

7.3.8 CHERI Exception Handling

As discussed in Section 7.2.3, we augmented the baseline CVA6 exception handling with capability awareness via SCRs. The `CSR Regfile` was extended to save and restore PCC in the `xepcc` and to redirect control flow using `xtcc`. These are the capability-extended versions of `xepc` and `xtvec`, respectively.

Capability Exception Reporting. Capability faults, such as tag violations or out-of-bounds accesses, raise a dedicated exception code ($0x1C$). Operand-specific diagnostics are reported via the `xcause` register. The `Branch`, `Load/Store`, and `CLU` units propagate these codes accordingly. As per CHERI specification, the MMU also raises specific exceptions during capability load/store operations ($0x1A$ and $0x1B$).

7.3.9 Tagged-Memory Support

CHERI requires a 1-bit tag per capability word to track capability validity. To support this, we adapted the CVA6 core `L1 Data Cache` and the interconnect interface to support tagged memory. At the `L1 Data Cache`, we reused the user-bit field to store the tag bit along with the capability data. We also configure the core to hold up to 2 capabilities per cache line (256-bits). At the interconnect interface, CVA6 utilizes the AXI protocol and an adapter to fetch memory according to the requested size (e.g., it can fetch an entire cache line by performing a burst operation). To propagate the tag bit along the interconnect interface, we adapted this AXI adapter by adding a 1-bit-wide user field to the read and write channels. Furthermore, the CVA6 AXI interface has a 64-bit data path. To maintain compatibility, we decided to keep the AXI data width at 64 bits and emulate fetches of capabilities as AXI bursts of 128 bits.

7.3.10 RVFI-DII Support

The RVFI-DII specification [602] extends the RISC-V RVFI interface for formal verification by enabling instruction injection and trace capture, bypassing speculative fetch and caches. This is essential for frameworks like TestRIG [390], the standard for RISC-V CHERI verification, which we use in our own validation flow (Section 7.5).

RVFI-DII Implementation. We implemented instruction injection by replacing the Instruction Cache (ICache) with a programmable injection engine interfaced via the top-level testbench. Early limitations included lack of support for compressed instructions and single-instruction injection blocking. These were later resolved through contributions from the Cambridge team[2], who extended the engine to support instruction injection and compressed instruction formats.

[2] https://github.com/zero-day-labs/cheri-cva6/pull/38

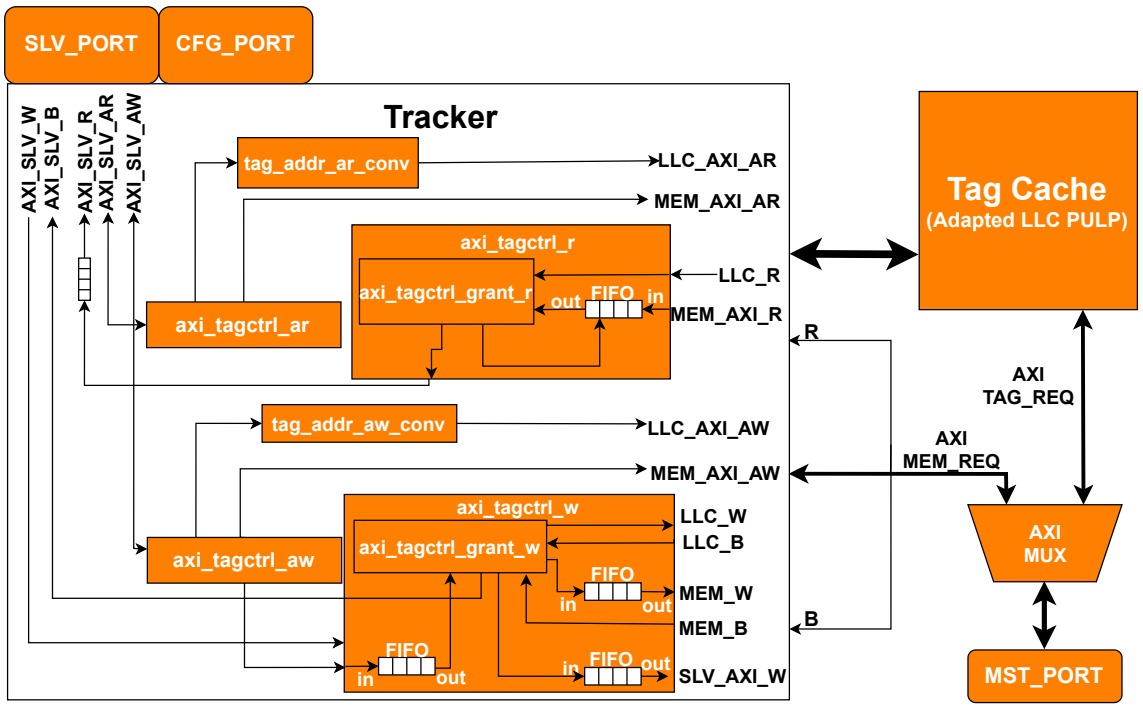


Figure 36: Tag Controller microarchitecture design high-level overview.

7.4 Tag Controller

Multiple approaches exist for managing capability tag storage in memory [434], including: (i) extending memory words to store tags alongside data (e.g., repurposing ECC bits); (ii) using dedicated tag memory and interfaces; and (iii) deploying a Tag Controller to emulate tagged memory. For our experimental setup, we adopt the third approach, which offers flexibility and aligns with prevailing practices in open-source CHERI systems. While an open-source Tag Controller exists from the Cambridge team, its Bluespec Verilog implementation and lack of documentation hinder integration and extensibility [3]. To address these limitations, we developed a new open-source Tag Controller in SystemVerilog, designed for seamless integration into CHERI systems using the AXI protocol and requiring no specialized or ECC-enabled memory, making it suitable for low- to mid-end platforms.

Tag Controller Overview. The Tag Controller emulates tagged memory by maintaining an in-memory bitmap of tag bits. It sits between the interconnect and main memory, intercepting the AXI transactions to retrieve both data and corresponding tag metadata. To reduce access latency, we include a configurable tag cache that buffers recent tag entries. Figure 36 illustrates the microarchitecture, composed of two main components: (i) the `Tracker`, responsible for coordinating memory-tag interactions, and (ii) the `Tag Cache`, which accelerates access to the tag bitmap. The following subsections describe each component in detail.

[3] https://github.com/CTSRD-CHERI/TagController

Tag Controller Tracker. The `Tracker` manages memory accesses and coordinates tag fetches from the `Tag Cache`. Figure 36 illustrates its microarchitecture. The AXI read and write channels are handled independently, although a shared interface module (`axi_tagctrl_ax`) processes both `AW` and `AR` channels. The transaction flow is as follows: (i) the Tracker registers the request based on access type (read/write), (ii) `axi_tagctrl_ax` initiates memory and tag cache transactions, converting data addresses to tag bit addresses, (iii) it issues transaction descriptors to the R/W handlers (`axi_tagctrl_r` and `axi_tagctrl_w`), (iv) for reads, data and tags are fetched and returned to the slave interface, and (v) for writes, data and tags are committed to memory and cache in parallel, completing only when both succeed. To improve throughput, the `Tracker` decouples slave and memory interfaces via FIFOs.

Tag Cache. The `Tag Cache` adapts logic from the PULP Platform's LLC design[4], incorporating minor changes for tag management. It comprises six main components: (i) `Read Unit`, (ii) `Write Unit`, (iii) `Refill Unit`, (iv) `Evict Unit`, (v) `Hit/Miss Detection Unit`, and (vi) `Data-Ways`. Communication is coordinated through request descriptors and decoupled using FIFOs and registers.

The Read and Write Units generate request descriptors and manage transaction flow. Interfaces were adapted to match the Tracker protocol. The `Data-Ways` handle all cache line accesses and were modified to support bit-level operations. The `Hit/Miss Detection Unit` checks tag presence and routes requests accordingly. The `Refill and Eviction Units` handle cache line management and leverage a Bloom filter to prevent concurrent operations on the same line.

7.5 Evaluation

In this section, we report the evaluation conducted on an experimental single-core CVA6-based CHERI SoC. Our evaluation focuses on verification and functional validation (Section 7.5.1), as well as performance and hardware resource overhead (Sections 7.5.3, 7.5.4, and 7.5.6). The system configurations under assessment, the methodology, benchmarks, and metrics are described below.

System and Tools. We ran the experiments on a CVA6 single-core SoC with a 32KiB Data Cache (DCache) and 16KiB ICache, deployed on the Genesys2 FPGA at 100MHz. The software stack comprises the following components: OpenSBI CHERI hybrid (version 0.9), FreeBSD (version 15.0), and CheriBSD (version 25.03). We compiled OpenSBI and CheriBSD using LLVM/Clang 15.0, which includes CHERI support. We compiled FreeBSD using LLVM/Clang 15.0. We synthesized the design and assessed the impact on FPGA hardware resources using Vivado version 2020.2. For the verification part, we used the TestRIG framework and the cycle-accurate simulator Verilator 5.0. To measure the execution time of each benchmark, we utilized the `time` utility provided by FreeBSD and CheriBSD, which measures the real, system, and user execution times.

Design Configurations. We evaluate two configurations. The baseline, `cva6h`, is a vanilla CVA6

[4] https://github.com/pulp-platform/axi_llc

core featuring a 32KiB DCache with 256-bit cache lines, a 16KiB ICache, hypervisor extension support, 32-entry BTB, 2-entry Return Address Stack (RAS), 128-entry BHT, and an enabled FPU. The extended configuration, `cva6h-cheri`, retains the baseline setup and augments it with CHERI support, tagged memory in the DCache, a TagController, and a 32 KiB TagCache (8-way set-associative, 128 lines per way, 32-byte lines, no hierarchical lookup).

Methodology. We focus our evaluation on execution cycles or functional performance. For each benchmark, we collected 1000 samples and filtered any outliers by computing the average of 95 percentile values. We normalized the results against the reference baseline configuration, where higher normalized values indicate lower performance (i.e., slowdown). For the baseline bar, we also included the absolute execution time. The baseline corresponds to the unmodified CVA6 core without CHERI and tagged memory, running the benchmark atop the FreeBSD OS (`freebsd-riscv64`). We ran the same benchmarks on top of the CheriBSD OS and `cva6h-cheri` design, compiled for legacy mode (`cheribsd-riscv64`) and pure-capability mode (`cheribsd-riscv64c`) and compared with the baseline results. All benchmarks were compiled with static linking and the default optimizations for each benchmark, $-O3$ or $-O2$. Each bar represents the total execution time (user time + system time). Each bar shows user time (opaque segment) and system time (shaded segment), with the total execution time given by their sum.

Benchmarks. We used two real-world application benchmark suites to assess the performance overhead: (i) MiBench Embedded Benchmark Suite [600] and (ii) San Diego Vision Benchmark [581] (SDVB). These suites incorporate a set of benchmarks widely used to assess the performance in embedded and mixed-criticality systems [56, 131]. The MiBench includes 35 application benchmarks divided into six different embedded market segments. We selected the automotive subset because it encompasses several memory-intensive benchmarks (`qsort`, `susan corners`, and `susan edges`) that stimulate multiple components across the memory hierarchy (e.g., tag controller, cache, and memory controller). San Diego Vision [581] comprises a complete set of vision applications that mimic real-world use-cases (e.g., motion tracking (`tracking`)) and support multiple datasets with different sizes. Finally, we also ran the CoreMark [599] benchmark in a baremetal environment (no OS). This synthetic benchmark focuses on evaluating the core performance using simple operations. We provide an online artifact with instructions on how to replicate our experiments on CoreMark, MiBench and SDV here[5].

7.5.1 Verification and Functional Validation

Unit-Level Validation. The first phase focuses on module-level testing using minimal testbenches built around the Verilator simulator and the GoogleTest framework. This step targets individual functional units of the CVA6 core, with emphasis on newly integrated CHERI extensions. Unit tests validate behavior under well-defined inputs and corner cases, ensuring correctness of capability manipulation, instruction decoding, and CSR/SCR logic. The implementation successfully passed in all of the unit tests.

[5]https://github.com/ninolomata/cva6h-cheri-benchmarks

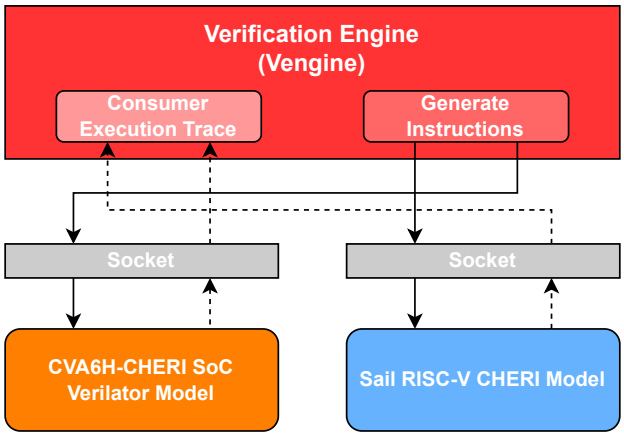


Figure 37: TestRIG verification setup overview.

Verification with TestRIG. The second phase employs the TestRIG framework to perform instruction-level verification of the CHERI extensions. Widely used in open-source CHERI implementations (e.g., Cheri-Toooba), TestRIG leverages the RVFI-DII interface (Section 7.3.10) to connect the core under test to a verification engine (`Vengine`). This engine injects CHERI instructions (randomized and model-derived from the CHERI Sail specification) into the core via RVFI-DII, enabling accurate verification against the ISA-level reference model. Figure 37 illustrates our TestRIG-based setup. Using TestRIG, we executed over 1000 randomly generated tests covering the entire RISC-V CHERI specification, with automated checks of the architectural state to ensure full compliance and functional correctness.

System-Level Validation with CheriBSD. The final phase validates end-to-end integration by deploying a single-core `cva6h-cheri` SoC to boot CheriBSD. This serves as a practical test of system stability, instruction compatibility, and tag propagation integrity under a real-world CHERI-aware operating system. Ultimately, the system successfully booted and ran CheriBSD, demonstrating correct functionality in a complete CHERI-aware software stack.

7.5.2 CoreMark Performance Results

We ran the CoreMark benchmark on both the `cva6h` and `cva6h-cheri` configurations in a bare-metal environment (i.e., without an operating system), executing CoreMark binaries compiled for legacy and pure-capability code. Table 26 presents the resulting CoreMark/MHz scores. The performance overhead of executing pure-capability code, when compared to the baseline legacy core, is negligible, approximately 6.3%. This result is consistent with overheads reported in the state-of-the-art (see Section 7.7) for this class of RISC-V CHERI-enabled cores.

Name	*cva6-legacy*	*cva6-purecap*
CoreMark/MHz	2.84	2.66

Table 26: CoreMark/MHz results for baseline CVA6 and CHERI-aware CVA6.

7.5.3 MiBench Performance Results

We evaluated the performance impact of CHERI on embedded workloads using the automotive subset of the MiBench benchmark suite. We executed each benchmark in three configurations: (i) a baseline setup using a vanilla CVA6 core running FreeBSD (`0-cva6-freebsd-riscv64`), (ii) the same binary executed on a CHERI-extended CVA6-based platform with CheriBSD (`1-cva6-cheribsd-riscv64`), and (iii) a pure-capability version of the benchmark compiled with CHERI Clang and executed on the same CHERI-enabled CVA6 platform (`2-cva6-cheribsd-riscv64c`). Therefore, comparing configurations (ii) and (i) captures the combined performance impact of moving to the CHERI-enabled platform and CheriBSD while retaining a non-pure-capability binary, whereas comparing configurations (iii) and (ii) quantifies the additional overhead introduced by executing a pure-capability binary. Figure 38 illustrates the normalized execution time overheads across all benchmarks and configurations. In configuration (ii), where the same binary is executed on CHERI-enabled hardware with CheriBSD, the performance overhead ranges from a minimum of 1.10x for `bitcount-large` to a maximum of 3.2x for `susanc-small`. The increase is predominantly observed in system execution time, suggesting that a significant fraction of the additional overhead originates from kernel-level execution in CheriBSD. Possible contributors include differences in system-call handling, memory management, and capability-related kernel operations; identifying the exact source would require more detailed profiling. At the user level, a modest performance increase is also observed in most benchmarks despite executing the same legacy binary. Since the application does not execute CHERI-specific instructions in this configuration, this overhead is likely associated with platform-level microarchitectural differences, particularly changes in the memory path introduced by the CHERI-enabled hardware. However, identifying the precise source of this overhead would require more detailed hardware performance-counter analysis. In contrast, configuration (iii) demonstrates at least a 10% performance slowdown compared to configuration (ii). Most of this overhead comes from the user execution, reflecting the overhead introduced by CHERI's capability checks, pointer manipulations, and tighter memory alignment constraints. The `bitcount` exhibits similar performance overhead results in pure-capability setup (ii) and legacy mode (iii). However, short-running benchmarks, such as `susanc-small` and `susane-small`, we observe an even more pronounced slowdown, primarily affecting system time execution. In Section 7.7, we provide a more in-depth analysis of these results compared to the state-of-the-art.

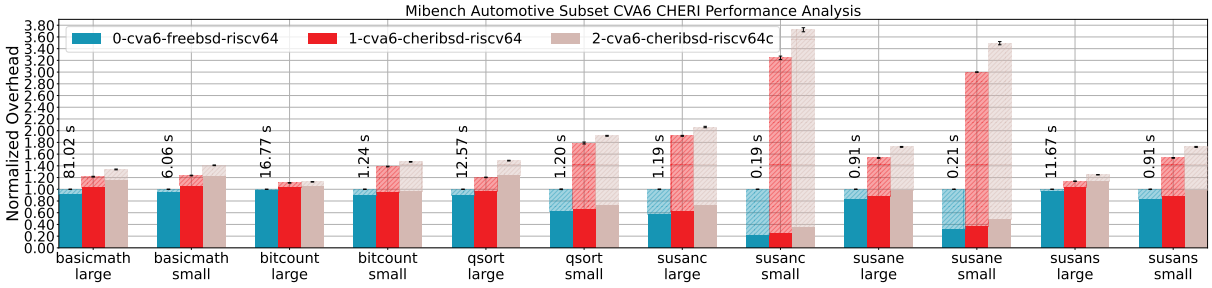


Figure 38: MiBench performance overhead results. Each bar is divided into two segments: the opaque segment denotes the proportion of total execution time spent in user-level, whereas the shaded segment denotes the proportion spent in kernel-level.

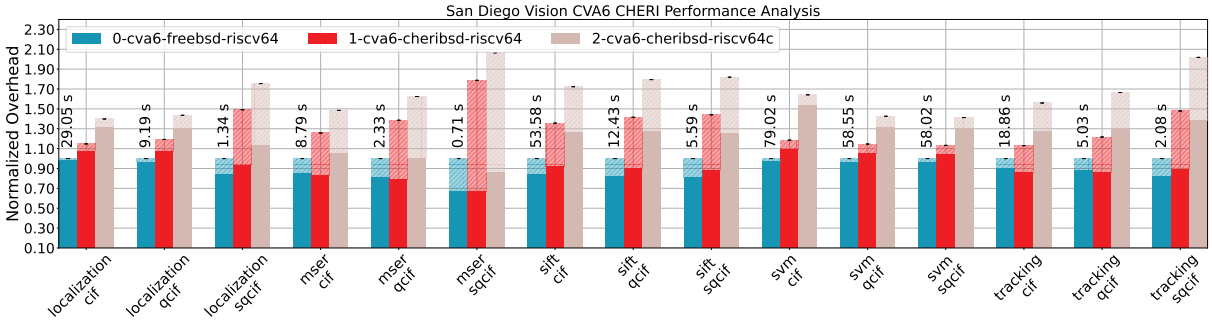


Figure 39: San Diego Vision performance overhead results. Each bar is divided into two segments: the opaque segment denotes the proportion of total execution time spent in user-level, whereas the shaded segment denotes the proportion spent in system-level.

7.5.4 San-Diego Vision Performance Results

For the San Diego Vision benchmarks, we used the same setups as defined in the MiBench Section 7.5.3. For each benchmark, we ran the algorithms at different resolutions (`sqcif`, `qcif`, `cif`), although other formats were available. However, due to the relatively low clock frequency of our platform, the execution time for higher resolutions becomes prohibitively long, making them impractical for our evaluation. Figure 39 shows the normalized execution time overheads across all benchmarks and configurations. In configuration (ii), compared with configuration (i), we observe a substantial performance overhead inflicted on the system time compared to the normal one, suggesting that like in MiBench benchmark, CheriBSD adds substantial runtime overhead compared to FreeBSD. The slowdown ranges from 1.1x (`svm-sqcif`) to a maximum of 1.7x (`mser-sqcif`). We can also observe that the overhead from configuration (ii) to (iii) remains steady at around 1.2x to 1.4x independently of the input image size. Finally, in configurations (ii) and (iii), most of the slowdown from the hybrid binary to the pure-capability stems from user-level execution, which is explained by the additional instructions required to handle CHERI capabilities.

Module	LUTs	FFs	BRAMs
CVA6	54598 / +54.3%	28137 / +36.3%	52 / +200%
Frontend	3270 / +18.47%	4445 / +21,32%	0 / +0%
Execute	23693 / +21,42%	10464 / +12,26%	0 / +0%
RegFile	1132 / +360%	1984 / +137%	0 / +0%
CSR Regfile	3245 / +31.8%	2509 / +33.3%	0 / +0%
CLU	0 / New (+2,200)	0 / New (+1,400)	0 / +0%
TagController	0 / New (+8235)	0 / New (+7762)	0 / New (32)
Branch Unit	612 / -73	0 / +0%	0 / +0%
LSU	12080 / +31.8%	7737 / +16.55%	0 / +0%
Cache	7977 / +79%	2638 / +39.6%	52 / +200%
Total	80065 / +53.9%	51291 / +43.2%	67 / +202%

Table 27: Post-Synthesis FPGA resource overhead of CVA6H-CHERI vs. Baseline. For modules not present in the baseline design, percentage overhead is not applicable; the absolute resource contribution of the newly introduced module is reported instead.

7.5.5 MiBench vs San Diego Vision

MiBench workloads generally show modest overhead in hybrid mode and moderate slowdowns in pure-capability mode, with higher impacts for memory-intensive workloads such as `susanc_small`. In contrast, several San Diego Vision workloads, particularly `mser_sqcif`, `sift_cif`, and `svm_cif`, exhibit significantly larger slowdowns in the pure-capability configuration, reflecting the higher sensitivity of pointer- and memory-intensive vision algorithms to CHERI's bounds and permission checks.

7.5.6 Hardware Overhead

Table 27 presents the utilization of post-synthesis FPGA resources of our CVA6H-CHERI implementation compared to the baseline CVA6 core. In general, we observe a LUT overhead of 53.9%, a FF increase of 43.2%, and a Block Random Access Memory (BRAM) usage increase of 202%. The most resource-intensive additions include the TagController (8235 LUTs, 7762 FFs, 32 BRAMs), which alone accounts for a significant share of the overall increase due to its role in managing out-of-band capability tag bits. Modules such as the LSU and RegFile also exhibit notable overheads as a result of the added logic for capability-aware memory operations and register file extensions. In contrast, the CLU adds a minimal footprint, confirming its lightweight nature for instruction-level capability manipulation. The cache subsystem shows an increase of 79% in LUT and 200% BRAM, primarily due to support for tagged memory and an expanded capacity width without optimizations. Interestingly, the Branch Unit shows a slight reduction in LUTs, due to streamlined handling of CHERI-aware control flow. We believe that with careful optimizations, for instance, at cache level, we could lower the overhead by at least 30%.

7.6 Discussion

Architectural and Microarchitectural Directions. This work implements the CHERI-RISC-V baseline specification version that served as the foundation for the standard now undergoing ratification, and is being updated in collaboration with the Cambridge team to maintain full alignment as the ratification process evolves. For instance, `Zcheri` [107] merges SCRs with existing RISC-V CSRs, enabling reuse of CVA6's CSR logic and potentially reducing hardware costs. At the microarchitectural level, we identify several optimizations. First, although our design predicts the full PCC, both bounds and permission checks could be deferred to later pipeline stages to reduce logic footprint, leveraging CVA6's in-order nature. Second, our Tag Controller may be enhanced to support a hierarchical tag cache, as demonstrated in prior work [434]. Finally, the current tag inspection mechanism brings both tag and data into the cache, which is suboptimal for tag-only operations (e.g., revocation). Supporting direct multi-tag inspection from main memory without data fetch could significantly improve efficiency.

Virtualization. We are investigating the use of CHERI in virtualization stacks. In MMU-based systems, virtualization typically enforces isolation at page granularity. We aim to assess how CHERI can complement or partially replace page-based mechanisms to improve determinism by: (i) enabling capability-mediated IPC, (ii) supporting fine-grained, revocable shared memory between VMs and the hypervisor, and (iii) enforcing compartmentalization within each VM and the hypervisor itself. For instance, hypervisor-managed VirtIO backends [136] (e.g., network devices) can be decomposed into per-VM compartments, each with bounded capabilities to its own `virtqueues`, thereby preventing cross-VM data leakage while enabling safe zero-copy buffers. We plan on leveraging CHERI capabilities to confine device backends, mediate shared memory, and mitigate the need for complex page-table manipulations.

In MMU-less systems, virtualization generally relies on an MPU to enforce physical memory isolation. However, MPUs provide only a small number of coarse-grained regions with strict alignment and size constraints, limiting scalability and flexibility. CHERI's architectural capabilities offer a more lightweight, granular and scalable alternative, particularly for mixed-criticality stacks that must meet real-time requirements under safety standards (e.g., `ISO 26262`). We are currently porting CHERI to the Bao hypervisor [118] and plan a design-space exploration comparing CHERI-based virtualization against traditional MPU-based approaches.

PPA Analysis. As part of our future work, we are conducting a PPA analysis of our CHERI-enabled CVA6 design using an industry-relevant technology node to quantify the practical cost of integrating CHERI into a high-performance RISC-V microarchitecture. This ensures that the reported area and power results are representative of realistic implementation conditions. Initial post-layout simulations suggest that, similar to prior virtualization work on CVA6, the area and power impact of CHERI-specific components, such as tag handling and capability-aware memory units, can be kept minimal with careful design.

Name	ISA	μArch	RVFI-DII	Hybrid	Evaluation		Lic.
					Performance	Area/LUTs	
CHERI Microcontrollers							
CHERI-Ibex [435]	RV32IMCB + M-mode + CHERIv8	in-order 2- or 3-stage	☑	☑	N/A	N/A	APv2
CHERI-Piccolo [436]	RV32IACDFM + M/U-mode + CHERIv8	3-stage in-order	☑	☑	CoreMark/MHz 4.1(10%)	VCU-118 FPGA +62%	BSD
CHERIoT-Ibex [437]	RV32IMCB + M/U-mode + CHERIoT ISA	2 or 3-stage in-order	☑	☐	CoreMark/MHz 1.811(13.18%)	TSMC 28nm HPC+ technology running at 300 MHz shows 2.15× the gates.	APv2
Proteus [439]	RV32IM + M-mode + CHERIv8 (no compressed capabilities)	5-stage in-order or super-scalar OoO	☐	☑	N/A	Zynq Ultrascale+ XCZU9EG FPGA +2.94%	MIT
CHERI Application Class Processors							
CHERI-Toooba [166]	RV64GC + CHERIv9	superscalar OoO	☑	☑	CoreMark/MHz 5 (-8%)	VCU-118 FPGA +45%	MIT & APv2
CHERI-Flute [165]	RV64GC + CHERIv8	5-stage in-order	☑	☑	CoreMark/MHz 3.6 (5.2%)	VCU-118 FPGA +50%	APv2
A730 CHERI [84]	RV64G RISC-V Standard (ZCheri)	dual-issue, in-order, 7-stage	☐	☑	N/A	+4%	C
CVA6 CHERI	**RV64GC + M/HS/VS/VU-modes + CHERIv9**	**6-stage in-order**	☑	☑	**CoreMark/MHz 2.84 (6.3%)**	**+53%LUTs and +43%FFs compared to the baseline targeting Genesys2 FPGA running at 100MHz**	**SP**

Table 28: Publicly available CHERI-enabled RISC-V CPU implementations, including: (i) CPU name with reference to the open-source repository; (ii) supported ISA; (iii) underlying microarchitecture; (iv) RVFI-DII support; (v) CHERI hybrid execution modes; (vi) CoreMark/MHz performance, area (e.g., FPGA LUTs or gate count), and power overheads; and (vii) licensing model (Apache v2, MIT, BSD, Solderpad Hardware, or Commercial).

7.7 Related Work

The RISC-V CHERI extensions are currently undergoing ratification. Nonetheless, multiple implementations of both commercial and open-source RISC-V cores have already integrated support for the upcoming specification (Zcheri). These include: (i) the commercial Codasip A730 processor [84], (ii) the open-source CVA6H-CHERI core, and (iii) CHERI-Toooba from the University of Cambridge [166].

Table 28 summarizes known CHERI-aware RISC-V cores (both commercial and open-source) and their key attributes. We categorize implementations into two groups: MCU and application-class processors. Most implementations follow the Cambridge version 9 CHERI-RISC-V specification or derivatives. The table also highlights the strong presence of permissive open-source licensing (Apache v2, Solderpad v2), reflecting the open and collaborative nature of the RISC-V ecosystem.

CHERI Microcontrollers. CHERIoT-Ibex [437] exemplifies a 32-bit pure-capability core tailored for IoT. It optimizes capability encoding for area and power efficiency and introduces temporal safety primitives. In contrast, hybrid designs such as Piccolo [436], Ibex [435], and Proteus [439] support mixed execution modes, enabling legacy software to run unmodified alongside CHERI-aware applications.

CHERI Application-Class Processors. CHERI-Flute [165] and CHERI-Toooba [166] are representative 64-bit RISC-V application-class processors that implement the CHERI version 8 ISA. Ongoing work on CHERI-Toooba targets compliance with the Zcheri specification. Our CVA6H-CHERI core is the first to implement CHERI version 9 along with the RISC-V hypervisor extension. On the commercial front, Codasip introduced the A730 processor [84], a CHERI-capable in-order core targeting mid-range applications. It supports the full Zcheri specification, and hybrid capability mode.

CHERI Performance Overhead. Existing performance studies predominantly rely on synthetic benchmarks (e.g., CoreMark/MHz), which fail to adequately stress critical microarchitectural components such as caches and branch predictors. In contrast, our evaluation leverages real-world workloads—specifically, MiBench and the San Diego Vision Suite, which provide a more representative characterization of CHERI system behavior. Table 28 compares reported CoreMark scores across various CHERI-enabled implementations. Most designs exhibit performance overheads in the range of 5.2% to 15% relative to their respective baselines, with the notable exception of CHERI-Toooba, which demonstrates improved performance due to a reduced branch misprediction rate [164]. Our results follow these trends, particularly for memory-light workloads such as `basicmath`. The observed performance of our CVA6-based implementation is consistent with cores of a similar class, such as CHERI-Flute, exhibiting an overhead of approximately 5%.

CHERI Area and Power Overhead. Comprehensive PPA studies remain scarce. CHERIoT-Ibex [437] and the A730 report less than 6% increases in gate count and power. FPGA-based results for CHERI-Flute and CHERI-Toooba show resource overheads ranging from 45% to 62% in LUTs and FFs. Prior studies [164] indicate that relative hardware overhead diminishes in larger cores, where unaffected units (e.g., FPUs) dilute the relative cost of CHERI logic. The CVA6-CHERI implementation follows this trend, incurring an overhead of only 50% additional LUTs and 43% registers relative to the baseline configuration.

7.8 Conclusion

This article presents our work on extending the CVA6 core with support for CHERI capabilities. We begin by providing a detailed overview of the CHERI architecture, followed by an in-depth discussion of our design and implementation. Central to our contributions is the development of a dedicated tag controller to manage capability tags in memory, which we integrated into a prototype CHERI-aware SoC. We conducted a comprehensive functional evaluation on an FPGA platform using three representative benchmark suites, MiBench, San Diego Vision, and CoreMark/MHz, to quantify performance overhead in terms of execution cycles. Our results demonstrate a performance overhead ranging from 1.1x up to 3.2x in the worst

case, with a resource utilization overhead of approximately 53.9% in FPGA LUTs and 43.2% in registers, respectively. All hardware designs, source code, and benchmark infrastructure have been open-sourced to promote reproducibility and support ongoing research in capability-based security for RISC-V architectures.

8

Bao-CHERI: A RISC-V Open-Source Hypervisor with CHERI Support

This chapter was submitted as:

Bruno Sá and Sandro Pinto. "*Bao-CHERI: A RISC-V Open-Source Hypervisor with CHERI Support*", Submitted to *IEEE Access*.

Abstract

Memory safety vulnerabilities remain a primary root cause of software exploits, particularly in embedded systems, where critical components such as hypervisors and firmware are predominantly implemented in memory-unsafe languages like C and C++. At the same time, embedded platforms are evolving towards increasingly complex and heterogeneous architectures. Combined with stringent SWaP-C constraints, this evolution is driving the consolidation of multiple software stacks with different criticality levels onto a single hardware platform, giving rise to MCSs. Virtualization has emerged as a practical foundation for such deployments, providing strong spatial isolation and predictable execution across partitions, as required by contemporary safety standards (e.g., ISO 26262). However, conventional isolation mechanisms primarily enforce protection across large system-level domains, such as processes or VMs, and offer limited support for fine-grained compartmentalization within a shared address space.

In this work, we present the integration of CHERI capability-based memory protection into the open-source Bao hypervisor, targeting the CHERI RISC-V ISA extension (version 9). To our knowledge, this is the first hypervisor to simultaneously support CHERI capabilities and the RISC-V hypervisor extension while operating in pure-capability mode, where all pointers are represented as capabilities. We also provide a detailed blueprint of the software modifications required to incorporate CHERI support into a RISC-V hypervisor. The integration effort and runtime overhead are evaluated on a single-core CHERI-enabled RISC-V CVA6 SoC FPGA prototype supporting both CHERI and the RISC-V hypervisor extension. The results indicate moderate overheads: (i) the hypervisor `.text` segment increases by 30.2%; (ii) runtime memory usage grows by approximately 280 B per vCPU; (iii) boot time increases by 20.09%; and (iv) interrupt latency increases by 13.43%. Additionally, application-level evaluation using the MiBench Automotive Benchmark Suite shows an average overhead of approximately 10% for pure-capability execution compared to legacy execution running atop Bao. Finally, Bao-CHERI is released as a fully open-source reference stack to support further research on CHERI-enabled virtualization.

8.1 Introduction

Embedded systems have been evolving rapidly, driven by a consolidation trend that aims to reduce SWaP-C through the integration of multiple software stacks with different criticality levels onto a single hardware platform [50]. In this context, SPHs [118, 131] have emerged as a practical foundation for mixed-criticality deployments in safety- and real-time-critical domains, as they provide strong spatial isolation and support predictable, non-interfering execution across partitions, critical for certification (e.g., ISO 26262) [561]. However, as system functionality grows, conventional isolation mechanisms (e.g., paging) are primarily designed to isolate large domains (e.g., kernels or user space applications) and provide limited support for fine-grained, intra-address space isolation. Simultaneously, critical software components, including hypervisors, firmware, OSs and device drivers, continue to be predominantly implemented in C/C++ due to their high performance and low-level control over hardware resources. As a result, memory-safety

vulnerabilities persist as the dominant root cause of system compromise [24, 25, 403].

In the embedded systems domain, design objectives typically prioritize deterministic execution and minimal runtime overhead. Consequently, many RTOSs adopt a monolithic single-address space model, in which all application tasks are executed within a shared protection domain. Alternatively, some systems rely on MPUs to enforce isolation between tasks (e.g., RISC-V SPMP [603]). Although this approach provides some isolation, it is fundamentally constrained by the limited number of protection regions and their coarse alignment requirements [604]. As a result, unrelated code and data objects are often forced to share the same region, weakening isolation guarantees. This makes compartmentalization brittle and highly sensitive to manual memory layout decisions.

CHERI [37, 394], as a hardware-software co-designed capability-based protection model, addresses this gap by enforcing deterministic memory safety and fine-grained compartmentalization via unforgeable, architecturally defined hardware capabilities, rendering many primitives of classical exploitation unrepresentable by construction. Previous work on CHERI, including CheriBSD [99], Linux Cherification [100], CHERIoT [87], and CheriOS [387], has demonstrated the benefits of deploying CHERI in feature-rich OSs and RTOSs. Although prior work [112, 443] introduced a CVA6 design with CHERI and hypervisor extension support and quantified its functional correctness along with its baseline hardware overheads, an important open question remains: *how and at what cost can CHERI be effectively leveraged within hypervisor-centric system architectures that dominate mixed-criticality embedded platforms?*

In this work, we report our effort to integrate CHERI support into the open-source Bao hypervisor that targets the CHERI RISC-V ISA extension (version 9) from Cambridge [394]. We also evaluate the porting effort by performing an overhead analysis on several metrics, including code size, runtime memory footprint, boot time, interrupt latency, and application performance. Our results show that enabling CHERI introduces moderate implementation and runtime overheads. At the source level, the port required the addition of 992 lines of code and modification of 353 lines, primarily localized within the architecture-specific layer, from an original codebase of approximately 17.9 KLoC. In terms of binary size, the hypervisor `.text` section increases by approximately 30.2%, while the `.bss` section grows by 59.4%, largely due to enlarged capability-aware runtime structures. Runtime memory overhead remains modest, increasing by roughly 1 KiB per core due to the expanded architectural register context. The boot time increases from 35.39 ms to 42.51 ms, corresponding to an overhead of 20.09%, mainly caused by the initialization of the capability relocation during system startup. Similarly, interrupt latency increases from 89.79μs to 101.85μs, reflecting a 13.43% increase due to the larger architectural state that must be saved and restored in traps. Finally, application-level performance, assessed using the MiBench Automotive Benchmark Suite, exhibits a moderate additional overhead of approximately 10% when pure-capability execution is enabled in a hosted environment. The resulting software stack forms a practical open-source reference platform that enables the investigation of: (i) CHERI-aware isolation models in partitioned virtualization; (ii) CHERI-based hardening of critical hypervisor code paths; and (iii) direct comparisons with traditional isolation mechanisms used in embedded consolidation (e.g., PMP/SPMP [603]).

To summarize, with this work we make the following contributions. First, we present, to the best

of our knowledge, the first SPH implementation to support the CHERI capability ISA extension, enabling capability-based memory protection in virtualized environments. In particular, we extend the open-source Bao hypervisor to support the CHERI RISC-V ISA extension and demonstrate its execution on a RISC-V platform featuring both CHERI and virtualization support (Section 8.2). Second, we provide a systematic blueprint of the software modifications required to support CHERI on the type-1 bare-metal Bao hypervisor. This blueprint documents the implementation effort required to port CHERI support to a hypervisor and may assist designers looking to integrate CHERI into their own virtualization platforms. Specifically, this includes modifications to the trap handling path, context switching mechanisms, capability-aware memory management, capability register save/restore procedures, and virtualization-related structures that must be adapted to correctly manage capability state across privilege levels (Section 8.3). Third, we conduct an evaluation of the integration effort and its impact on system overhead. In particular, we analyze implementation complexity, code size growth, memory footprint, boot-time overhead, interrupt latency, and application-level performance using representative embedded benchmarks (Section 8.4). These results provide initial insight into the trade-offs associated with enabling CHERI across the full virtualization system stack. Finally, we outline several future research directions enabled by this work, including the use of CHERI capabilities to compartmentalize hypervisor services and internal subsystems to improve fault isolation and security within the hypervisor itself (Section 8.5). We also make our CHERI-enabled Bao hypervisor port available as an open-source artifact to support reproducibility and further research within the RISC-V and CHERI ecosystems.[1]

8.2 System Architecture

Figure 40 depicts an example architecture of a CHERI-aware virtualization system stack on RISC-V. The system stack is composed of: (i) the hardware platform, (ii) Bao at the hypervisor layer, and (iii) one or more VMs running an OS, RTOS, or bare-metal application, with CHERI enabled or disabled at the guest.

8.2.1 Platform Requirements

We assume a single-core or multicore platform featuring: (i) a RISC-V core (e.g., CVA6); (ii) support for the RISC-V Hypervisor extension with a two-stage MMU (e.g., `Sv39x4` on RV64); (iii) CHERI-RISC-V support compliant with CHERI version 9, including both integer and pure-capability execution modes, along with a merged capability register file; and (iv) tagged-memory support for capability tags.

8.2.2 Hypervisor Layer

At the hypervisor layer, Bao is compiled in *pure-capability* mode, meaning that all pointers are represented as CHERI capabilities with explicitly defined bounds and permissions. Bao is an open-source,

[1] https://github.com/ninolomata/bao-hypervisor/tree/feat/cheri

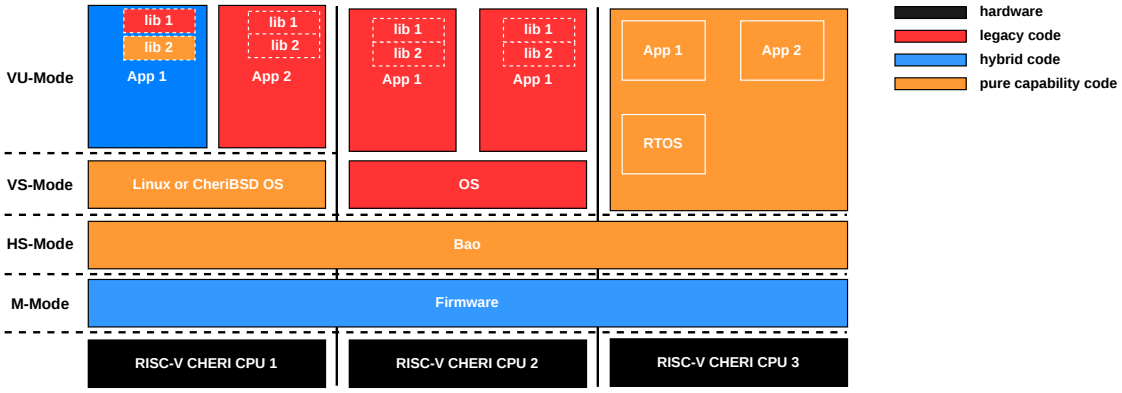


Figure 40: Example architecture of a tri-core RISC-V platform running Bao with two CHERI-enabled VMs and one legacy VM. One core runs a rich OS (Linux or CheriBSD) with capability-enforced user-level libraries, while the second core runs a RTOS or a bare-metal applications, and the third runs a legacy system without CHERI support.

type-1 hypervisor that uses static partitioning to divide system resources (e.g., memory, CPUs, devices) among VMs at boot time. Concretely, Bao implements a minimal software layer that: (i) creates and boots VMs by statically assigning resources, (ii) manages second-stage page tables to translate guest physical addresses to host physical addresses and isolate VMs from one another; (iii) provides intra-VM communication channels via shared memory; (iv) routes traps and exceptions (including firmware calls via `ecall`); and (v) supports device passthrough and interrupt routing (e.g., direct interrupt injection to guest VMs when feasible).

Our design adopts a hybrid protection model, in which Bao enforces coarse-grained isolation between VMs using the MMU, while CHERI capabilities provide fine-grained protection and object-level memory safety within the hypervisor itself (and optionally within guest systems). Although Bao relies on statically allocated data structures to manage VMs and CPUs, CHERI can still be leveraged to harden critical code paths and private per-CPU structures. For example, Bao maintains per-CPU private control structures, per-VM metadata (e.g., register state, memory layout, CPU assignments and configuration data), and per-vCPU execution state. Encapsulating these structures within bounded capabilities can improve fault containment and reduce the likelihood that memory-safety violations (e.g., stack overflows, pointer corruption, or invalid memory accesses) compromise critical hypervisor state.

8.2.3 VM Layer

At the guest layer (Figure 40), we support three distinct execution models: (i) a legacy VM without CHERI support, (ii) a hybrid VM that combines pure-capability components with conventional code, and (iii) a fully pure-capability software stack. The guest can operate either with or without an MMU, depending on the design requirements of the target OS/RTOS. In this configuration, Bao does not enforce CHERI semantics on the guest; rather, a guest may selectively adopt CHERI mechanisms to compartmentalize its applications, OS services, and libraries. This approach preserves compatibility with legacy systems, facilitates

the incremental introduction of CHERI within the guest environment, and enables future extensions such as capability-protected shared-memory IPC channels between VMs.

8.3 Bao-CHERI RISC-V Implementation

This section details the modifications required in Bao to support CHERI on RISC-V. Although this implementation targets CHERI version 9 [394], the design choices are described in terms of the architectural mechanisms that remain relevant to ongoing RISC-V `Zcheri` standardization. The objective is to provide a blueprint of the minimal set of changes necessary at the hypervisor–guest boundary to enable the boot, execution, and isolation of capability-aware software stacks. Concretely, the Bao modifications target four subsystems: (i) the boot process, (ii) the trap entry and exit, (iii) the MMIO and peripherals (e.g., interrupt controller and UART), and (iv) the memory management logic.

8.3.1 Boot Process

Boot is a critical integration point when enabling CHERI, as it establishes a well-defined architectural state and the initial capability root from which all subsequent authority is derived. Figure 41 presents a high-level overview of the Bao CHERI boot sequence, where blocks highlighted in red denote CHERI-specific additions or modifications relative to the original Bao boot flow. Several steps reuse the original Bao logic but are internally adapted to operate on CHERI capabilities; however, these are not highlighted to maintain clarity of the figure.

Boot Assumptions and Entry State. Bao relies on the preceding boot stage (firmware) to establish a minimal execution contract. First, the firmware is required to supply an initial PCC capability with sufficient authority (in terms of bounds and permissions) to execute the early boot code and to derive the initial code and data capabilities required by Bao. Second, Bao assumes that the initial boot path executes in `pure-capability mode`. If the core starts executing in integer mode, the reset handler transitions to capability mode by configuring the PCC state and transferring control via a capability-based control-flow transfer (e.g., using `cspecialr` + `csetflags` + `cjr`). ***Early Trap Vector Setup (`stcc`).*** To enable the safe handling of early exceptions, Bao installs a trap entry capability immediately upon entry. Under CHERI, the integer register `stvec` is replaced by its capability counterpart `stcc`, which stores a capability referencing the trap vector entry point. The boot code constructs a capability for the trap entry routine (e.g., `_hyp_trap_vector`) by deriving it from PCC (using `cllc`/`csetaddr`) and writes this capability into `stcc` via `cspecialw`. This procedure guarantees that trap entry is capability-safe from the earliest stages of execution.

Known Supervisor State. Following installation of the trap vector, the boot path initializes the hart into a well-defined supervisor state by disabling interrupts and virtual memory translation (e.g., clearing

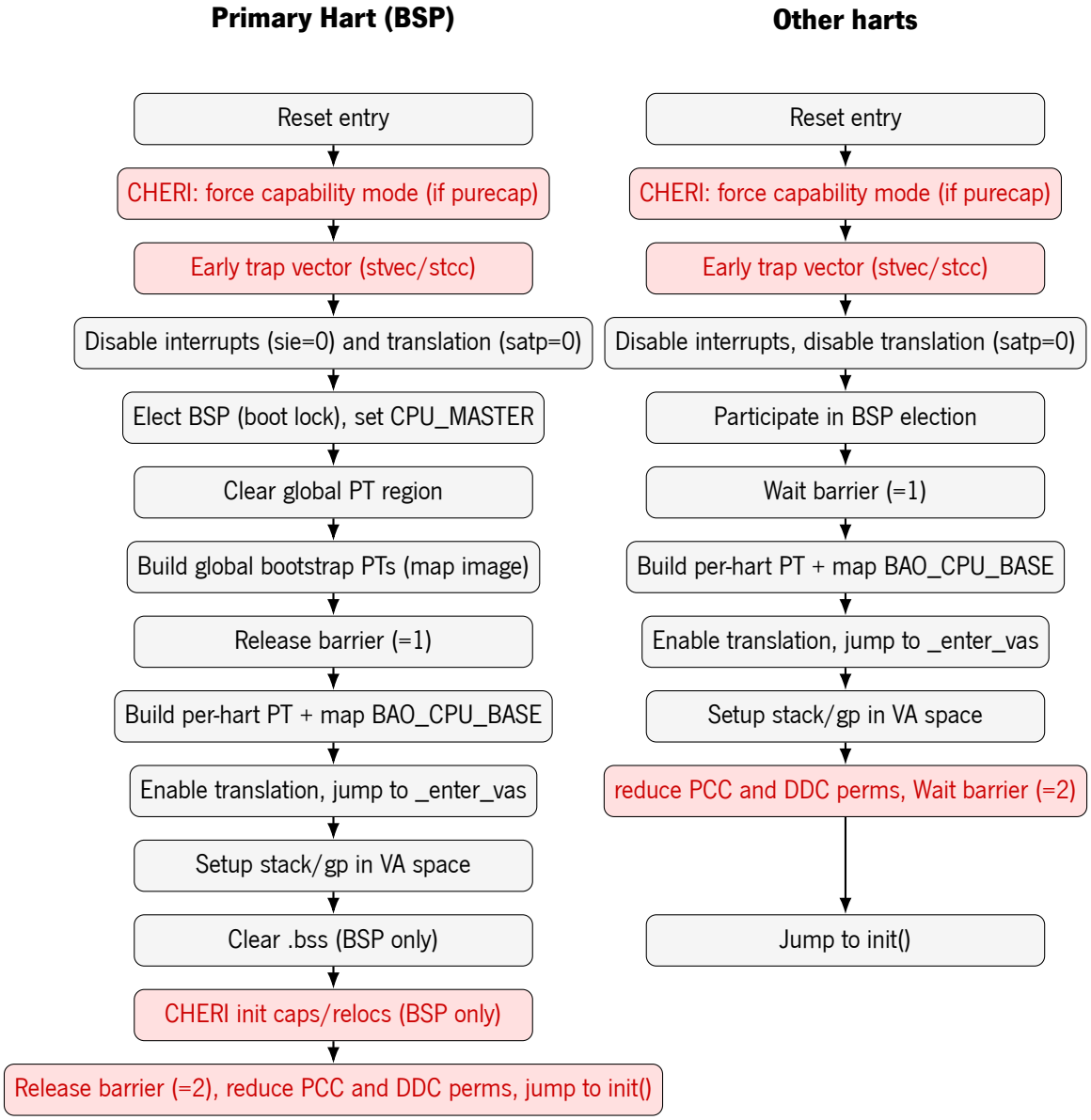


Figure 41: Bao-CHERI high-level boot and initialization sequence with BSP vs secondary-hart responsibilities. Steps highlighted in red correspond to CHERI-specific operations and modifications to the original boot flow.

`sstatus`, `sie`, `sip`, and `satp`). This ensures deterministic early execution while page tables and per-hart state are being established.

Bootstrap Synchronization and BSP Election. Bao adheres to the standard RISC-V bootstrap procedure by electing a bootstrap hart (`CPU_MASTER`) responsible for performing global initialization, while the remaining harts block on a synchronization barrier. This execution path must remain capability-safe because it implements shared-memory synchronization and acquires locks using atomic operations. In pure-capability mode, lock locations are represented as capabilities (e.g., obtained via `cllc`), and atomic instructions employ capability-based addressing variants to avoid reverting to unsafe integer-based pointer

arithmetic.

Bootstrap Page Tables and Temporary Physical Mapping. Bao constructs a minimal set of page tables (e.g., `Sv39` on RV64) to transition from an initial physical mapping to Bao's final virtual address-space layout, typically using identity mappings during early bootstrap. In pure-capability mode, all pointer arithmetic on page-table structures is performed using capability offset operations (e.g., `cincoffset`) instead of integer `add` instructions. To limit code duplication, the bootstrap code abstracts these differences behind helper macros (e.g., `LOAD`, `STORE`, `LOAD_ADDR`, and `PTE_PTR`), which map conventional integer-based operations (`ld`, `sd`, `la`, and `add`) to their CHERI-capability counterparts (`cld`, `csd`, `cllc`, and `cincoffset`) when compiling in pure-capability mode.

Transition to Virtual Address Space. After constructing the page tables, Bao programs `satp`, executes `sfence.vma` to invalidate stale address-translation state, and transfers control to the virtual entry point (e.g., `_enter_vas`). Under CHERI, this control transfer is implemented using a capability-aware jump (`cjr`) so as to preserve capability metadata and authority across the transition from physical to virtual addressing. Once operating in the virtual address space, Bao reinstalls the trap vector using virtual addresses by rewriting `stcc`.

Per-hart Stack Initialization and Global Pointer Handling. Bao allocates a private stack for each hart within a CPU-local memory region. In pure-capability mode, the stack pointer is represented as a *bounded capability*: the runtime derives a stack capability from a trusted data capability, computes the per-hart base address, and applies `csetbounds` to confine authority to the corresponding stack region. Because pure-capability code does not rely on the global pointer register, Bao clears `cgp` (e.g., `cgp = cnull`) and instead utilizes the compiler-generated capability table (*captable*) for accessing global data.

Capability Initialization (`DDC` and Capability Relocations). Prior to transferring control to the main initialization routine, Bao configures the CHERI capability environment. First, Bao clears `ddc` (sets it to `cnull`) to preclude inadvertent use of legacy *DDC-relative* load and store instructions within hypervisor code. Second, Bao invokes the compiler-provided capability-relocation initialization routine (e.g., `cheri_init_caps_reloc`), which materializes static code and data capabilities as specified by the compiler-generated relocation metadata (including bounds, permissions, and optional sealing attributes). This procedure initializes the global capabilities consumed by the C code that runs in pure-capability mode.

Reducing PCC Authority. The reset path typically commences with a highly privileged PCC capability. To enforce the principle of least privilege, Bao restricts PCC permissions to the hypervisor code permission set (e.g., `CHERI_HYP_CODE_PERMS`) and can additionally tighten the PCC bounds to encompass only the Bao text segment.

Instruction and CSR Mapping Summary. Table 29 summarizes the conventional RISC-V instructions and CSRs operations in Bao's boot code that require CHERI-specific replacements or adaptations for pure-capability execution.

Category	Conventional RISC-V	CHERI Pure-Capability
Load / Store	`ld`, `sd`	`cld`, `csd`
Atomic Instructions	`lr.w`, `sc.w`	`clr.w`, `csc.w`
Address Materialization	`la`	`cllc`
Pointer Arithmetic	`add`	`cincoffset`
Trap Vector Setup	`stvec`	`stcc`
CSR Access	`csrw`, `csrr`	`cspecialw`, `cspecialr`

Table 29: Instruction and CSR mapping required for transitioning Bao boot code from conventional RISC-V to CHERI pure-capability execution.

8.3.2 Trap Entry/Exit and Context Switching (CHERI Deltas)

A fundamental integration point between CHERI and Bao is the trap entry/exit pathway, as it determines (i) the trusted boundary between guest and hypervisor execution and (ii) the exact machine state that must be preserved across privilege transitions. In a pure-capability system, this state comprises not only the integer registers and conventional CSRs, but also the capability registers and capability-specific SCRs. Consequently, our implementation introduces a CHERI-aware trap vector that saves and restores the complete guest context, switches to a spatially bounded hypervisor stack, and re-establishes the hypervisor capability environment prior to dispatching to synchronous exception or interrupt handlers.

Capability-aware scratch register and trapframe base. In the baseline RISC-V Bao hypervisor, the trap handler employs `sscratch` to maintain either a pointer to the per-CPU trapframe or a stack anchor. Under CHERI, this pointer must instead be represented as a capability; consequently, we rely on the capability CSR `sscratchc`. Upon trap entry, the prologue exchanges a designated scratch capability register (e.g., `c31`) with `sscratchc` using `cspecialrw`. This mechanism yields a spatially bounded, unforgeable capability referencing the guest context save area (i.e., the trapframe base), allowing trap-entry code to access the saved context without relying on unrestricted pointers.

Saving guest architectural state. After obtaining the trapframe base capability, the trap prologue spills the guest general-purpose register file into the trapframe. In pure-capability mode, these stores are capability-based and are commonly encapsulated via macros (e.g., `STORE_REG`). Beyond the integer register state, the handler preserves virtualization-relevant privileged state necessary for correct guest resumption, including `hstatus` and `sstatus`. Crucially, in pure-capability mode the return program counter must be restored as a capability. Accordingly, we save `sepcc`, the capability counterpart of `sepc`. When CHERI is enabled, we additionally preserve the guest `ddc`, as it forms part of the guest's architectural protection context for legacy, non-capability-aware addressing.

Switching to a bounded hypervisor stack. Once the guest state has been fully saved, the handler transitions execution to hypervisor-private memory prior to invoking any C code. In the pure-capability execution path, a fresh stack capability is derived from a trusted root data capability, offset to the per-CPU

stack region, and spatially constrained via `csetbounds`. The handler then adjusts the stack pointer to the top of this bounded stack and clears `cgp`. This design ensures that, even in the presence of a trap handler vulnerability, writable authority is structurally confined to the stack bounds (and any other explicitly held capabilities), rather than being implicitly inherited from an unbounded stack pointer.

Trap dispatch under pure-capability execution. After establishing the hypervisor execution context, the handler reads `scause` and dispatches to either the synchronous exception handler or the interrupt handler. In pure-capability mode, all indirect control-flow transfers employ capability-aware call/jump instructions (e.g., constructing a code capability via `cllc` and transferring control via `cjalr`). This preserves CHERI control-flow provenance and mitigates common "jump-to-integer-address" gadget patterns that are characteristic of traditional control-flow hijacking attacks.

Restoring guest state and re-entering the VM. The return path restores `hstatus` and `sstatus`, as well as the guest `ddc` and guest `sepcc`. A CHERI-specific constraint is that the guest return program counter must be written as a capability (via `cspecialw sepcc, …`), ensuring that the subsequent `sret` instruction resumes guest execution with the correct authority and spatial bounds. Finally, the general-purpose registers are restored from the trapframe, including the scratch register that serves as the trapframe base pointer.

8.3.3 Other Modifications

In addition to the boot process and trap entry/exit paths, which are predominantly implemented in assembly and require a systematic replacement of integer-based addressing with CHERI capability semantics, further Bao subsystems were updated to preserve capability safety. In particular, any component that derives pointers from raw physical or virtual addresses was refactored to operate on bounded capabilities constructed from trusted roots.

MMIO and Peripherals. Interactions with MMIO devices (e.g., Platform-Level Interrupt Controller (PLIC) and UART) were adapted to follow capability-based addressing. Instead of directly using raw physical addresses to access device registers, the hypervisor constructs bounded capabilities at runtime from trusted root capabilities. These capabilities restrict access to the intended MMIO regions and enforce spatial safety when interacting with device registers.

Memory Management. The memory management subsystem was also adapted to support capability-aware operation. The raw-address computations responsible for calculating addresses, doing page-table manipulation, and setting up memory regions, were instead replaced by safe capabilities with appropriate bounds and permissions. These changes ensure that the memory accesses performed by the hypervisor remain within authorized regions.

8.4 Evaluation

In this section, we present an evaluation of the Bao-CHERI port based on four primary metrics: (i) code size and memory footprint, (ii) boot time, (iii) interrupt latency, and (iv) performance overhead. The following subsections detail the hardware platform, software toolchain and software stack, as well as the experimental methodology employed to extract the results.

Tools and System Stack. We carried out our experiments on a single-core CVA6 SoC emulated on a Genesys2 FPGA operating at 100 MHz. The software stack used for the experiments includes: (i) OpenSBI CHERI hybrid (version 0.9), (ii) CheriBSD (version 25.03), a (iii) CHERI-enabled fork of the original Bao hypervisor, and (iv) a CHERI Baremetal application. All components were compiled with LLVM/Clang 15.0. For performance evaluation, we relied on the time utility provided by CheriBSD to measure the execution time of each benchmark.

Design Configurations. The design under evaluation is a CVA6 core configured with a 32 KiB DCache using 256-bit cache lines, a 16 KiB instruction cache ICache, support for the hypervisor extension v1.0, support for version 9 of the CHERI RISC-V extension, a 32-entry BTB, a 2-entry RAS, a 128-entry BHT, and an enabled FPU. In addition, it includes a TagController, and a 24 MiB TagCache that is 8-way set-associative with 128 lines and without hierarchical lookup.

Benchmarks. For the performance evaluation, we used the MiBench Embedded Benchmark Suite, a widely adopted collection of benchmarks derived from real-world applications, to quantify performance overheads in real-time embedded systems and MCS [118, 131]. MiBench comprises 35 application benchmarks organized into six categories representative of distinct embedded market segments. In this work, we focus on the automotive subset, as it includes several memory-intensive benchmarks (e.g., `qsort`, `susan corners`, and `susan edges`) that exercise multiple components of the memory hierarchy, such as the tag controller, cache, and memory controller.

8.4.1 Code Size and Memory Footprint

Methodology. To measure the impact of CHERI on code size and memory footprint, we compare the original Bao hypervisor against its CHERI-enabled version along two dimensions: (i) source-level code changes, and (ii) static and dynamic memory usage. For the source-level analysis, we use the `cloc` tool to compute the number of source lines of code (SLoC) per directory and programming language, and to classify changes into unchanged, modified, added, and removed lines. For the static binary analysis, we inspect the final ELF binaries using `readelf` and `objdump` to extract the size of the main sections (i.e., `.boot`, `.text`, `.rodata`, `.data`, and `.bss`) on a per-directory basis. This approach enables us to attibute code and data growth to specific components of the system. For the dynamic memory analysis, we inspect per-vCPU runtime data structures that are affected by CHERI. In particular, we analyze the private per-vCPU architecture-specific register context structure (`struct arch_regs`), which is extended to

store capability-sized registers and system registers (SCRs in CHERI).

Directory	Language	Same	Modified	Added	Removed
arch/riscv	C	2719	70	409	27
	C/C++ Header	1170	44	247	2
	Assembly	170	183	179	0
	Make	59	6	9	0
core	C	2857	41	66	1
	C/C++ Header	847	5	19	0
	Make	23	0	0	0
platform	C	577	1	33	0
	C/C++ Header	592	0	8	0
	Make	142	1	12	1
lib	C	309	0	0	0
	C/C++ Header	296	0	0	0
	Make	3	0	0	0
Total	–	**17520**	**353**	**992**	**31**

Table 30: Source-level code changes by directory and language (SLoC).

Results and Analysis. Tables 30, 31 and 32 summarize the impact of CHERI on code size, source-level complexity, and runtime memory footprint. From a source-level perspective, enabling CHERI results in a moderate increase in code complexity. Across all directories, 992 lines were added and 353 modified, primarily concentrated in the `arch/riscv` directory, which reflects the architectural changes required to support capability-aware execution and register management. The rest of the system, including core functionality and platform support, remains largely unchanged.

From a static binary perspective, CHERI significantly increases the size of key sections. The `.text` segment grows by 30.2%, due to capability-aware instruction sequences and runtime checks. The `.bss` section increases by 59.4%, reflecting larger statically allocated structures, and the `.data` segment increases by 26.5%. Additionally, CHERI introduces a `.capreloc` section of 8.6 KiB, which encodes relocation metadata to initialize global and static capabilities at runtime. This section is processed during boot time and contributes to the initial memory image.

In terms of runtime memory, CHERI increases the size of Bao's per-vCPU state. Bao allocates each VM's internal structure at runtime as a contiguous block containing a `struct vm` followed by an array of `struct vcpu` objects. Each `struct vcpu` contains the saved architectural register state in `struct arch_regs`. In the CHERI configuration, this register state expands because general-purpose register slots are represented as capability-width values, and extra fields such as the default data capability (`ddc`) and alignment padding are added. Consequently, `struct arch_regs` increases from 312 bytes to 568 bytes (+82.1%). This increase propagates directly to the size of each `struct vcpu`, adding roughly 300 bytes of persistent runtime memory per-vCPU after accounting for CHERI-related padding and alignment. On a configuration with one vCPU per physical core, a quad-core platform therefore incurs approximately 1.2 KiB of additional persistent runtime memory.

Directory	Section	Vanilla-Bao	Bao-CHERI	Overhead
		Size (Bytes)	Size (Bytes)	(%)
arch/riscv	.boot	952	1228	+29.0%
	.text	13184	18284	+38.7%
	.rodata	72	112	+55.6%
	.data	428	1306	+205.1%
	.bss	40	176	+340.0%
core	.boot	0	0	–
	.text	27456	35208	+28.2%
	.rodata	0	0	–
	.data	400	512	+28.0%
	.bss	17684	30212	+70.8%
lib	.boot	0	0	–
	.text	2236	2352	+5.2%
	.rodata	264	132	-50.0%
	.data	0	0	–
	.bss	0	0	–
platform	.boot	0	0	–
	.text	20	24	+20.0%
	.rodata	0	0	–
	.data	664	704	+6.0%
	.bss	0	0	–
Total	.boot	952	1228	+29.0%
	.text	42896	55868	+30.2%
	.rodata	4440	5164	+16.3%
	.data	1948	2464	+26.5%
	.bss	21161	33745	+59.4%

Table 31: Binary section sizes by directory (bytes): Vanilla Bao vs Bao-CHERI.

Component	Vanilla Bao (Bytes)	Bao-CHERI (Bytes)	Absolute Increase	Overhead (%)
`struct arch_regs`	312	568	+256	+82.1%
`struct vcpu_arch`	40	64	+24	+60.0%
`struct vcpu` (partial)	–	–	≥280	–
Per-vCPU total overhead	–	–	≈280–320 B/core	–

Table 32: Per-vCPU runtime memory overhead: Vanilla Bao vs Bao-CHERI.

8.4.2 Boot Time Overhead

Methodology. To evaluate the boot time overhead introduced by CHERI, we implemented a minimal bare-metal benchmark application and instrumented the system to measure the elapsed time from the first instruction executed by the Bao hypervisor until the application starts and produces its first observable output (i.e., prints a message on the console). Boot time is measured using the architectural timer, which provides a resolution of 40 ns. The benchmark application is small, with a code footprint of approximately 2 KiB, in order to minimize application-level effects and isolate the overhead of the hypervisor boot sequence. We collect 100 independent boot samples for each configuration and report the average boot time. We carried out the evaluation using the following: (i) the unmodified Bao hypervisor

(`vanilla-bao`), and (ii) the CHERI-enabled Bao version running in pure-capability mode (`cheri-bao`), with the application also compiled as a pure-capability binary.

Name	avg	std-dv	min	max
vanilla-bao	35394201	3126	35386000	35397760
cheri-bao	42505314 (+20.09%)	2439 (−21.95%)	42502240 (+20.11%)	42510840 (+20.09%)

Table 33: Boot time overhead (ns).

Results and Analysis. Table 33 reports the boot-time measurements for both configurations. In the baseline setup, `vanilla-bao` exhibits an average boot time of $35.39\,\mathrm{ms}$, with low variance across runs. When CHERI support is enabled, the average boot time increases to $42.51\,\mathrm{ms}$, corresponding to an overhead of approximately 20.09%. The minimum and maximum boot times also increase by a similar margin, indicating that the overhead is systematic and not due to outliers. This overhead is primarily due to the additional initialization work performed by the hypervisor under CHERI. In particular, the CHERI-enabled boot sequence must initialize the capability table (`captable`) by processing the `__cap_relocs` metadata section, which involves constructing valid capabilities with appropriate bounds, permissions, and base addresses for all global objects. This process requires iterating over relocation entries and performing multiple capability derivation and bounds-setting operations, significantly extending the critical boot path. Moreover, the initialization of capability-aware execution contexts requires saving and restoring a larger architectural state, including capability registers and associated metadata. Collectively, these factors explain the approximately 20% boot-time overhead observed in the CHERI-enabled configuration.

8.4.3 Interrupt Latency

Methodology. To assess interrupt latency overhead, we implemented a minimal bare-metal benchmark application that programs the architectural timer to raise periodic interrupts at fixed intervals of 100 ms. Since the expected firing time of each interrupt is precisely known, we measure latency as the difference between the programmed wall-clock timestamp and the actual timestamp at which the interrupt handler begins execution. The timer offers a resolution of 40 ns. We collect 1000 samples and compute the average latency. Experiments were conducted under two configurations: (i) the unmodified Bao hypervisor without CHERI extensions (`vanilla-bao`), and (ii) the CHERI-enabled Bao version compiled in pure-capability mode (`cheri-bao`), along with the bare-metal application also compiled as a pure-capability binary.

Name	avg	std-dv	min	max
vanilla-bao	89794.44	2831.33	86680	167000
cheri-bao	101852.04 (+13.43%)	2918.65 (+3.08%)	99760 (+15.09%)	188720 (+13.01%)

Table 34: Interrupt latency overhead (ns).

Results and Analysis. Table 34 summarizes the results for each configuration, reporting the average, standard deviation, minimum, and maximum interrupt latency values. For `cheri-bao`, we additionally report the relative overhead compared to `vanilla-bao`. In the baseline configuration, the average interrupt latency is $89.79\ \mu$s, with a standard deviation of $2.83\ \mu$s, a minimum of $86.68\ \mu$s, and a maximum of $167.00\ \mu$s. When CHERI support is enabled (`cheri-bao`), the average latency increases to $101.85\ \mu$s, reflecting a 13.43% overhead. The minimum and maximum latencies also increase to $99.76\ \mu$s and $188.72\ \mu$s, corresponding to increases of 15.09% and 13.01%, respectively. The standard deviation rises slightly by 3.08%, indicating that the temporal variability remains low in the measured runs. This overhead comes from two main factors. First, the trap handling path in Bao becomes more expensive under CHERI, as saving and restoring execution context involves a larger set of registers, including capability registers with associated metadata. Second, pure-capability code generated by the CHERI compiler toolchain exhibits increased code size and alignment constraints, leading to a larger instruction and data footprint. This footprint places greater pressure on the memory hierarchy, particularly the instruction and data caches, both due to code size and the frequent load/store operations on capabilities, which are about twice as large than conventional pointers. These effects likely explain the increase in average and tail latencies.

8.4.4 Performance Overhead

Methodology. For performance evaluation, we focus on measuring the overhead in terms of execution time. For each benchmark, we collected 1000 samples and pruned outliers by averaging the values within the 95th percentile. Figure 42 illustrates the normalized execution time overheads (i.e., slowdown) across the MiBench automotive benchmarks and all configurations, relative to a reference baseline configuration (`cva6-cheribsd-riscv64`), with absolute execution times shown above each bar. Each bar in the figures represents the total execution time (user time + system time). The user time is shown as an opaque segment and the system time as a shaded segment; their sum yields the total execution time. The figure compares four configurations: hybrid-capability CheriBSD without Bao (`cva6-cheribsd-riscv64`), pure-capability CheriBSD without Bao (`cva6-cheribsd-riscv64c`), and their corresponding hosted variants atop the Bao hypervisor (`cva6-bao-cheribsd-riscv64` and `cva6-bao-cheribsd-riscv64c`). All benchmarks are statically linked and compiled with their default optimization level, either `-O3` or `-O2`.

Results and Analysis. First, we compare configurations (0) and (1), which quantify the cost of transitioning from hybrid to pure-capability execution in a virtualized environment. In both configurations, the kernel operates in pure-capability mode, whereas user space switches from hybrid to pure-capability execution in configuration (1). As shown in Figure 42, this transition induces a modest performance degradation across all benchmarks, typically below 10%. For compute-bound workloads, such as `bitcount-small`

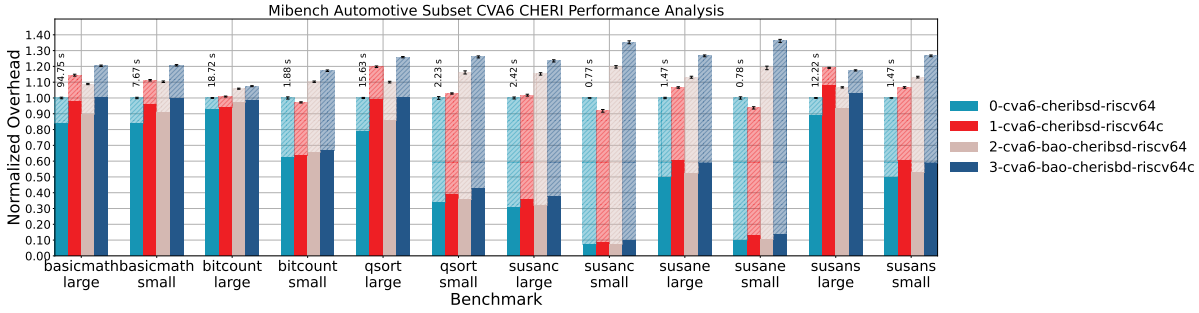


Figure 42: MiBench performance overhead results. Each bar is divided into two segments: the opaque segment denotes the proportion of total execution time spent in user-level, whereas the shaded segment denotes the proportion spent in kernel-level.

and `qsort-small`, the normalized overhead remains close to the legacy execution, whereas memory-intensive workloads, such as `susans-large` and `susane-large`, exhibit moderately higher slowdowns. This behavior is consistent with the additional costs associated with pure-capability code generation, enlarged pointer representations, and the requirement for the kernel to maintain and validate capability state during system calls and exception handling, even in the absence of virtualization.

Next, comparing configurations (0) and (2) isolates the overhead introduced by Bao hosting while preserving hybrid user execution. As illustrated by the hatched bars in Figure 42, enabling Bao leads to a relatively uniform slowdown across benchmarks, with normalized overheads generally in the range of 5%–20%. Compute-bound benchmarks such as `bitcount-large` incur comparatively low overhead (approximately 5%), whereas system- and memory-intensive workloads such as `susanc-small` and `susane-small` experience overheads approaching 20%. The performance penalty is more pronounced in system time than in user-mode execution. These effects can be attributed to virtualization overheads, including timer virtualization, trap handling, and second-stage address translation.

Furthermore, comparing configurations (1) and (3) captures the combined impact of Bao virtualization and pure-capability execution. In this setting, the slowdown becomes more substantial, with normalized overheads frequently exceeding 40% for benchmarks such as `susanc-small` and `susane-small`, which have relatively small absolute execution times and are therefore more sensitive to increases in system time, representing the worst cases among the evaluated benchmarks.

Finally, comparing configurations (2) and (3) isolates the incremental cost of CHERI in a hosted environment. This transition introduces an additional normalized overhead of approximately 10% on average, with the impact most evident in user-time execution. This increase reflects the cost of pure-capability user code executing atop an already virtualized kernel, including capability-aware system calls, increased register pressure due to capability registers, and an enlarged memory footprint. It is also important to note that the evaluated platform does not incorporate microarchitectural optimizations for virtualization, such as a second-stage TLB, an L2 TLB, or hardware-assisted timer virtualization. Consequently, the reported results should be interpreted as a conservative estimate of the overhead on this unoptimized platform.

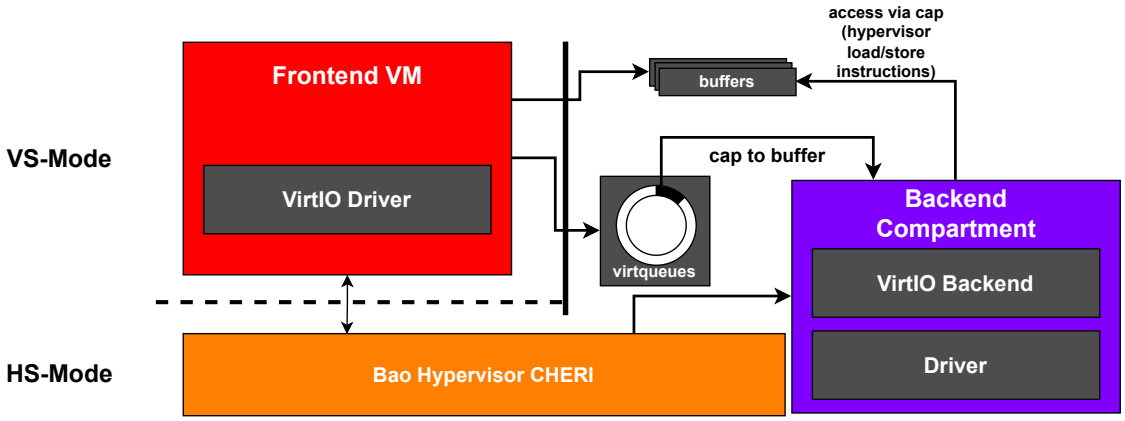


Figure 43: Architecture overview of CHERI-enabled VirtIO design in Bao with a compartmentalized backend service and capability-mediated shared memory communication.

8.5 Exploratory Design Directions

In this section, we outline a set of exploratory design directions that extend our current architecture. Each subsection describes how CHERI's capability-based model can enhance specific aspects of virtualization and system software for MCS.

8.5.1 Compartmentalization of Hypervisor Services: VirtIO Use-Case

A recurring challenge in virtualization is the secure and efficient sharing of a single physical device among multiple VMs. NICs, for example, are frequently multiplexed across several guest systems. A widely adopted technique to facilitate device sharing is paravirtualization, which decouples the guest device driver from the physical device backend and is becoming the de facto approach nowadays in automotive applications. VirtIO [137] defines a standardized interface for such paravirtualized devices based on a split-driver model. In this model, the device logic is partitioned into two components: (i) a frontend driver that runs within the guest and (ii) a corresponding backend component that lives in the host or hypervisor, often implemented in user space with frameworks such as QEMU [128]. Communication between the frontend and backend is performed through shared-memory ring buffers, referred to as `virtqueues`, which consist of a descriptor table, an available ring, and a used ring.

VirtIO Overview. To issue an I/O request, the frontend enqueues one or more descriptors into the available ring. Each descriptor specifies the physical address of the guest and the length of an associated memory buffer. The backend then directly accesses the referenced memory region, executes the requested operation, and signals completion by updating the used ring. This architecture enables zero-copy data transfers and reduces the number of VM exits relative to full device emulation, thus achieving near-native I/O performance. However, this performance optimization is based on a strong trust assumption that the backend is granted broad access to the guest memory. Consequently, the backend operates with ambient authority, and there is no fine-grained, hardware-enforced isolation to restrict its access to specific

memory regions. For example, in Xen [115], paravirtualized I/O backends that execute in the privileged service domain (dom0) can freely access guest memory through shared buffers and event channels. As a result, I/O backends substantially enlarge the TCB.

Bao VirtIO Support. Bao has also been extended to support VirtIO in BiRtIO [136]. BiRtIO confines the VirtIO backend to a dedicated VM, distinct from both the frontend VM and the hypervisor. Since VMs in Bao are strictly isolated and cannot directly share arbitrary memory, BiRtIO uses bounce buffers and explicitly shared memory regions to mediate communication. Bounce buffers are intermediate memory regions that are used to copy data between the frontend and backend. All I/O buffers are transferred from guest-private memory to shared memory regions and then copied back, introducing overhead in terms of memory consumption, latency, and CPU utilization, particularly in high-throughput workloads.

CHERI for VirtIO. We propose enhancing the BiRtIO architecture using the capability-based memory protection model supported by CHERI. Figure 43 illustrates our design for RISC-V. Instead of implementing the backend as a separate VM, we propose to deploy it as a compartmentalized CHERI application service executing in the hypervisor layer within the same address space. This avoids costly VM transitions and reduces the implementation complexity of the backend to that of a regular task application. In this model, the VirtIO frontend and backend communicate using shared memory regions represented by capabilities. Rather than sharing raw memory addresses, the frontend shares capability-wrapped descriptors in the `virtqueue`, each granting bounded, revocable access to a buffer. The backend operates using only these capabilities, with hardware-enforced checks ensuring that it can access only the explicitly authorized memory regions. To access frontend memory via capabilities and without breaking isolation, we propose two mechanisms:

1. **Capability-aware hypervisor load/store:** RISC-V provides hypervisor memory access instructions (e.g., `HLVX`), which respect guest translation and protection rules. A capability-aware variant of these instructions could allow the backend to dereference shared capabilities using the guest's memory view while maintaining CHERI's spatial safety guarantees. The advantage of this model is that the CHERI extension already provides these instructions and would require no modifications to the original ISA.

2. **Physical capabilities:** As stated in [387, 394], physical capabilities bypass virtual memory translation and directly refer to physical memory. These could be delegated to backends for zero-copy access without relying on bounce buffers or page remapping. This model requires new extensions to the original ISA, and different interpretations of capabilities at the hardware level.

Challenges. This CHERI-enhanced design enables zero-copy data sharing, strong isolation, and reduced backend latency, while preserving Bao's core principles of minimality and static partitioning. However, it introduces new challenges, particularly around capability revocation. Once a capability is shared with the backend, mechanisms are needed to revoke or restrict its use when the frontend VM exits or reclaims its

buffers. This remains an open research problem requiring further investigation into revocation semantics across compartment boundaries.

8.5.2 CHERI as an Alternative to MMU-less Virtualization

The consolidation of real-time services in embedded systems often suffers from the unpredictable latency introduced by MMU-based virtualization. Frequent and unpredictable page table walks caused by TLB misses and cache-dependent behavior can result in nondeterministic timing, which is unacceptable for real-time workloads. Although MMU-based approaches offer virtually unlimited address spaces and flexible memory management, their reliance on complex hardware structures makes it difficult to provide strong temporal guarantees. Therefore, recent approaches propose MMU-less virtualization [141] based on two-stage MPU configurations. In this model, the guest OS configures a first-stage MPU to isolate applications, while the hypervisor enforces global isolation between VMs and the hypervisor itself through a second-stage MPU. This architectural approach, already implemented in ARM Cortex-R platforms (e.g., Armv8-R) [138], facilitates more stringent control over temporal predictability and spatial isolation. An analogous mechanism is currently being incorporated into the RISC-V ecosystem through the standardization of the SPMP [603] extension for hypervisor.

Limitations. Although MPU-based virtualization provides strong timing predictability, it imposes several fundamental structural constraints. First, MPUs typically support only a small, fixed number of protection entries, which restricts the number of concurrently protected regions (e.g., a maximum of 192 entries in RISC-V SPMP for hypervisor RV64) and often necessitates memory clustering or frequent reconfiguration in more dynamic execution contexts. Second, guests must be explicitly aware of the physical memory layout a priori, which complicates portability, lacks flexibility, and complicates dynamic memory management. Consequently, while MPU-based designs improve temporal determinism, they do so at the cost of flexibility and scalability.

CHERI as a Secure Partitioning Mechanism. We propose investigating CHERI as an alternative isolation substrate for MMU-less systems. In contrast to MMU-based virtualization, CHERI avoids the timing variability associated with address translation, such as TLB misses and page-table walks. As summarized in Table 35, this enables object-level isolation rather than page- or region-level protection, without reliance on page tables, fixed-size regions, or centralized memory metadata. At boot time, the hypervisor can provision capabilities that precisely define the memory and I/O authority of each VM or real-time partition, thus eliminating ambient authority and avoiding static region mappings.

Relative to MPU-based approaches, CHERI removes the inherent limitation on the number of protected regions and supports dynamic memory allocation and controlled sharing through explicit capability passing. In contrast to MMU-based virtualization, CHERI avoids nondeterministic memory translation based on TLB and page-table walks, improving timing predictability while reducing the reconfiguration overheads associated with context switches. More importantly, the CHERI protection model is decentralized and does

not rely on a global memory configuration. Capabilities explicitly encode authority across domains and are continuously checked in hardware. This design vision aligns with ongoing efforts such as CHERIoT, which demonstrate that CHERI can be used to compartmentalize real-time operating systems by enforcing fine-grained isolation between software components in the absence of an MMU. CHERIoT [87] provides empirical evidence that capability-based protection can replace traditional MPU-based designs while still preserving real-time guarantees. This supports the characterization in Table 35 of CHERI as offering both high timing predictability and support for flexible memory-protection schemes. Building on these results, CHERI-based virtualization offers a promising foundation for hybrid MMU-less hypervisors that combine strong spatial isolation and hardware-enforced memory safety.

Property	**2-stage-MMU**	**2-stage-MPU**	**CHERI-based**
Isolation granularity	Page-level	Region-level	Object-level
Hardware structures	Page tables, TLB	MPU/PMP entries	Capabilities
Timing predictability	Affected by TLB/page walks	High	High
Dynamic memory support	High	Limited	High
Number of regions	Virtually unlimited	Very limited	Virtually unlimited
Guest memory awareness	Not required	Required	Not required
Pointer-level spatial memory safety	No	No	Yes
Fine-grained sharing	Difficult	Very limited	Native
Reconfiguration overhead	High (TLB, walks)	High (reprogramming the MPU)	Low (capability passing)

Table 35: Comparison of isolation mechanisms for embedded virtualization.

8.6 Conclusion

This article presented the integration of CHERI capability-based memory protection into the Bao hypervisor targeting RISC-V platforms. We first described the protection models made possible by combining CHERI with the Bao hypervisor in the RISC-V ecosystem, as well as the software modifications required to support pure-capability execution within Bao and its guest VMs. To quantify the integration effort and runtime overhead, we conducted an experimental evaluation on an early CHERI-enabled RISC-V CVA6 FPGA platform that supports both the RISC-V hypervisor extension and CHERI. Our evaluation considers several dimensions, including code changes, binary size, runtime memory footprint, boot-time overhead, interrupt latency, and application-level performance using representative embedded benchmarks. The results indicate moderate overheads: (i) the hypervisor `.text` segment size increases by approximately 30.2%; (ii) runtime memory usage grows by about 280 B per-vCPU; (iii) boot time increases by 20.09%; and (iv) interrupt latency increases by 13.43%. Application-level performance, assessed using the MiBench Automotive Benchmark Suite, exhibits an average overhead of approximately 10% for pure-capability execution in a hosted environment. Overall, the findings demonstrate that the CHERI approach can be integrated into a SPH with acceptable overheads while providing stronger memory safety and isolation guarantees. Furthermore, we release the CHERI-enabled Bao hypervisor as a fully open-source reference platform to

support future research on leveraging capabilities to further compartmentalize hypervisor services, thereby enhancing fault isolation and security in MCS.

Bibliography

[1] J. Perez-Cerrolaza et al. “Virtualization Technology for Dependable Embedded Systems: A Survey for Transportation and Industrial Domains”. In: *ACM Computing Surveys* (2025). doi: 10.1145/3765736 (cit. on pp. 2, 15, 17).

[2] M. R. Kabir and S. Ray. “Virtualization for Automotive Safety and Security Exploration”. In: *IEEE 16th Dallas Circuits and Systems Conference (DCAS)*. 2023. doi: 10.1109/DCAS57389.2023.10130221 (cit. on pp. 2, 15, 17).

[3] F. A. Administration. *Use of Virtual Machines in Avionics Systems and Related Assurance Concerns*. Tech. rep. TC-19/22. U.S. Department of Transportation, 2019 (cit. on pp. 2, 15, 17).

[4] M. Mounir. *Using Hypervisor for Infotainment and AUTOSAR Consolidation on a Single ECU*. 2021 (cit. on pp. 2, 15, 17).

[5] T. A. Henzinger and J. Sifakis. “The Embedded Systems Design Challenge”. In: *FM 2006: Formal Methods*. 2006. doi: 10.1007/11813040_1 (cit. on p. 2).

[6] J. P. Cerrolaza et al. “Multi-core Devices for Safety-critical Systems: A Survey”. In: *ACM Computing Surveys* (2020). doi: 10.1145/3398665 (cit. on p. 3).

[7] S. Pinto et al. “Virtualization on TrustZone-Enabled Microcontrollers? Voilà!” In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019. doi: 10.1109/RTAS.2019.00032 (cit. on pp. 3, 111).

[8] D. Cerdeira et al. “ReZone: Disarming TrustZone with TEE Privilege Reduction”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022 (cit. on pp. 3, 54, 111, 136).

[9] G. Heiser. “Virtualizing Embedded Systems - Why Bother?” In: *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2011 (cit. on pp. 3, 111).

[10] M. Bechtel and H. Yun. “Exploiting DRAM Bank Mapping and HugePages for Effective Denial-of-Service Attacks on Shared Cache in Multicore”. In: *Proceedings of the Symposium on Hot Topics in the Science of Security (HotSoS)*. New York, NY, USA, 2020 (cit. on p. 3).

[11] H. Lefeuvre et al. “SoK: Software Compartmentalization”. In: *IEEE Symposium on Security and Privacy (SP)*. 2025. doi: 10.1109/SP61157.2025.00075 (cit. on p. 3).

[12] M. Broy et al. “Engineering Automotive Software”. In: *Proceedings of the IEEE* (2007). doi: 10.1109/JPROC.2006.888386 (cit. on p. 3).

[13] K. Koscher et al. “Experimental Security Analysis of a Modern Automobile”. In: *IEEE Symposium on Security and Privacy*. 2010. doi: 10.1109/SP.2010.34 (cit. on p. 3).

[14] S. Checkoway et al. “Comprehensive Experimental Analyses of Automotive Attack Surfaces”. In: *20th USENIX Security Symposium (USENIX Security 11)*. 2011 (cit. on p. 3).

[15] M. Masmano et al. “XtratuM: A Hypervisor for Safety Critical Embedded Systems”. In: *11th Real-Time Linux Workshop*. 2009 (cit. on pp. 3, 17, 49, 60).

[16] G. Gracioli et al. “Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms”. In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. 2019. doi: 10.4230/LIPIcs.ECRTS.2019.27 (cit. on pp. 3, 123).

[17] T. Kloda et al. “Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019. doi: 10.1109/RTAS.2019.00009 (cit. on pp. 3, 18).

[18] W. Jo and I. Ahmed. “Oops, it halted again: exploiting PLC memory for fun and profit in industrial control systems”. In: *Proceedings of the 19th USENIX Conference on Offensive Technologies*. 2025 (cit. on p. 3).

[19] L. Davi et al. “Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection”. In: *23rd USENIX Security Symposium*. 2014 (cit. on p. 3).

[20] E. A. Lee. “Cyber Physical Systems: Design Challenges”. In: *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008. doi: 10.1109/ISORC.2008.25 (cit. on p. 3).

[21] International Organization for Standardization. *ISO 26262: Road vehicles - Functional safety*. Accessed on: 08-02-2026. 2018. url: https://www.iso.org/standard/68383.html (cit. on pp. 3, 19, 48).

[22] L. Bilge and T. Dumitraş. “Before We Knew It: An Empirical Study of Zero-Day Attacks in The Real World”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 2012. doi: 10.1145/2382196.2382284 (cit. on p. 3).

[23] Verizon Business. *2025 Data Breach Investigations Report*. Tech. rep. Verizon, 2025 (cit. on p. 3).

[24] M. Miller. “Trends and Challenges in the Vulnerability Mitigation Landscape”. In: *13th USENIX Workshop on Offensive Technologies*. 2019 (cit. on pp. 4, 22, 73, 159).

[25] Paul Kehrer. *2021 in Memory Unsafety - Apple's Operating Systems*. 2021. url: https://langui.sh/2021/12/13/apple-memory-safety/ (cit. on pp. 4, 22, 73, 159).

[26] J. V. Stoep and A. Rebert. *Eliminating Memory Safety Vulnerabilities at the Source*. Google Online Security Blog. 2024. url: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html (cit. on pp. 4, 32).

[27] Cppcheck. *Cppcheck: A Tool for Static C/C++ Code Analysis*. Open-source static analysis tool. url: https://cppcheck.sourceforge.io/ (cit. on pp. 4, 39, 57).

[28] A. Bessey et al. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World". In: *Communications of the ACM* (2010). doi: 10.1145/1646353.1646374 (cit. on pp. 4, 39, 57).

[29] A. Chou et al. "An Empirical Study of Operating Systems Errors". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. 2001. doi: 10.1145/502034.502042 (cit. on pp. 4, 39).

[30] Y. Zhang et al. "Debloating Address Sanitizer". In: *31st USENIX Security Symposium*. 2022 (cit. on pp. 4, 36, 74).

[31] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. "WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 2014. doi: 10.1145/2581122.2544147 (cit. on pp. 4, 36, 41).

[32] T. C. Team. *Hardware-assisted AddressSanitizer Design Documentation*. url: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html (cit. on pp. 4, 36, 41).

[33] K. Serebryany. "ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety". In: *login Usenix Magazine* (2019) (cit. on pp. 4, 37, 41, 57).

[34] RISC-V Memory Tagging Task Group. *RISC-V Memory Tagging Extension (Zimt)*. Tech. rep. Draft Version v0.3.0. RISC-V International, 2026 (cit. on pp. 4, 37, 41).

[35] Y. Li et al. "PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022. doi: 10.1145/3548606.3560598 (cit. on pp. 4, 37, 41).

[36] K. Hohentanner, P. Zieris, and J. Horsch. "CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++". In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 2023. doi: 10.1145/3555776.3577635 (cit. on pp. 4, 37).

[37] J. Woodruff, R. N. Watson, D. Chisnall, et al. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014. isbn: 978-1-4799-4394-4 978-1-4799-4396-8. doi: 10.1109/ISCA.2014.6853201 (cit. on pp. 4, 7, 8, 23, 24, 37, 41, 74, 75, 78, 81, 136, 138, 159).

[38] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018. isbn: 1593278284 (cit. on pp. 4, 32, 34, 41, 55, 74).

[39] J. D. Ichbiah. “Preliminary Ada reference manual”. In: *SIGPLAN Notices* (1979) (cit. on pp. 4, 34, 35, 41, 55).

[40] The Linux Kernel Community. *Rust - The Linux Kernel Documentation*. Accessed on: 09-02-2025. url: `https://docs.kernel.org/rust/index.html` (cit. on pp. 4, 35).

[41] O. d. Moor et al. “Keynote Address: .QL for Source Code Analysis”. In: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2007. doi: `10.1109/SCAM.2007.31` (cit. on pp. 4, 39).

[42] P. Cuoq et al. “Frama-C: A Software Analysis Perspective”. In: *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*. 2012. doi: `10.1007/978-3-642-33826-7_16` (cit. on pp. 4, 38, 41).

[43] K. Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. 2012 (cit. on pp. 4, 35, 40, 41, 55).

[44] T. C. Team. *UndefinedBehaviorSanitizer (UBSan)*. Accessed on: 28-10-2025. url: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html` (cit. on pp. 4, 36, 41, 55).

[45] E. Stepanov and K. Serebryany. “MemorySanitizer: Fast detector of uninitialized memory use in C++”. In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015. doi: `10.1109/CGO.2015.7054186` (cit. on pp. 4, 36, 57).

[46] A. 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*. 2013. doi: `10.1145/2508859.2516713` (cit. on pp. 4, 7, 37, 79, 81, 87).

[47] G. J. Duck and R. H. C. Yap. “Heap Bounds Protection with Low Fat Pointers”. In: *Proceedings of the 25th International Conference on Compiler Construction*. 2016. doi: `10.1145/2892208.2892212` (cit. on pp. 4, 37).

[48] G. J. Duck, R. H. C. Yap, and L. Cavallaro. “Stack Bounds Protection with Low Fat Pointers”. In: *Network and Distributed System Security Symposium (NDSS)*. 2017 (cit. on pp. 4, 37).

[49] J. Zhou, J. Criswell, and M. Hicks. “Fat Pointers for Temporal Memory Safety of C”. In: *Proceedings of the ACM on Programming Languages* (2023). doi: `10.1145/3586038` (cit. on pp. 4, 37).

[50] G. Heiser. “The Role of Virtualization in Embedded Systems”. In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. 2008. doi: `10.1145/1435458.1435461` (cit. on pp. 5, 15, 17, 158).

[51] G. Heiser. *The seL4 Microkernel - An Introduction*. Tech. rep. The seL4 Foundation, 2020. url: https://sel4.systems/About/seL4-whitepaper.pdf (cit. on pp. 5, 7, 21, 47, 49, 59, 60, 62, 74, 75, 81, 82, 98).

[52] L4Re.org. *L4Re Operating System Framework - Interface and Usage Documentation*. 2025. url: https://l4re.org/doc/l4re_intro.html (cit. on pp. 5, 21, 48, 75, 81, 82).

[53] K. Lampka et al. "Safety Certification with the Open Source Microkernel-Based Operating System L4Re". In: *Proceedings of 41st International Conference on Computer Safety, Reliability, and Security*. 2022. doi: 10.1007/978-3-031-14835-4_3 (cit. on pp. 5, 47, 48, 59).

[54] R. Uhlig et al. "Intel virtualization technology". In: *Computer* (2005). doi: 10.1109/MC.2005.163 (cit. on pp. 5, 16, 111).

[55] Arm Ltd. *Isolation using virtualization in the Secure world Secure world software architecture on Armv8.4*. 2018 (cit. on pp. 5, 6, 111).

[56] B. Sá, J. Martins, and S. Pinto. "A First Look at RISC-V Virtualization From an Embedded Systems Perspective". In: *IEEE Transactions on Computing* (2022). doi: 10.1109/TC.2021.3124320 (cit. on pp. 5, 6, 16, 112, 114, 116, 123, 132, 133, 138, 148).

[57] C. Dall et al. "ARM Virtualization: Performance and Architectural Implications". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016. doi: 10.1109/ISCA.2016.35 (cit. on pp. 5, 16).

[58] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys* (2019). doi: 10.1145/3291047 (cit. on pp. 5, 6, 53, 60, 108).

[59] N. C. Will and C. A. Maziero. "Intel Software Guard Extensions Applications: A Survey". In: *ACM Computing Surveys* (2023). doi: 10.1145/3593021 (cit. on pp. 5, 108).

[60] D. Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *Proceedings of the 15th European Conference on Computer Systems*. 2020. doi: 10.1145/3342195.3387532 (cit. on pp. 5, 54, 108).

[61] D. Cerdeira et al. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems". In: *IEEE Symposium on Security and Privacy (S&P)*. 2020. doi: 10.1109/SP40000.2020.00061 (cit. on pp. 6, 54).

[62] M. Lipp et al. "Armageddon: Cache Attacks on Mobile Devices". In: *Proceedings of the 25th USENIX Security Symposium*. 2016, pp. 549–564 (cit. on p. 6).

[63] X. Li et al. "Design and Verification of the Arm Confidential Compute Architecture". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022 (cit. on pp. 6, 55, 60).

[64] R. Sahita et al. “CoVE: Towards Confidential Computing on RISC-V Platforms”. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023. doi: 10.1145/3587135.3592168 (cit. on pp. 6, 55, 60).

[65] W. T. Group. *WorldGuard Specification*. Tech. rep. RISC-V Internacional, 2023. url: https://lists.riscv.org/g/security/attachment/711/0/worldguard_rvia_spec-v0.4.pdf (cit. on pp. 6, 55, 60, 107).

[66] J. B. Dennis and E. C. Van Horn. “Programming Semantics for Multiprogrammed Computations”. In: *Communications of the ACM* (1966). doi: 10.1145/365230.365252 (cit. on pp. 6, 20, 76, 138).

[67] W. Wulf et al. “HYDRA: the kernel of a multiprocessor operating system”. In: *Communications of the ACM* (1974). doi: 10.1145/355616.364017 (cit. on pp. 7, 21, 22, 81, 82, 138).

[68] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS: a fast capability system”. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. 1999. doi: 10.1145/319151.319163 (cit. on pp. 7, 21, 22, 47, 81, 82, 138).

[69] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984 (cit. on pp. 7, 20, 22, 76, 80).

[70] B. W. Lampson. “Dynamic Protection Structures”. In: *Proceedings of Fall Joint Computer Conference*. 1969. doi: 10.1145/1478559.1478563 (cit. on pp. 7, 21, 76, 138).

[71] N. Hardy. “The Confused Deputy: (Or Why Capabilities Might Have Been Invented)”. In: *SIGOPS Operating Systems Review* (1988). doi: 10.1145/54289.871709 (cit. on pp. 7, 21, 74, 138).

[72] N. Hardy. “KeyKOS Architecture”. In: *SIGOPS Operating Systems Review* (1985). doi: 10.1145/858336.858337 (cit. on pp. 7, 21, 22, 76).

[73] G. Klein et al. “seL4: formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 2009. doi: 10.1145/1629575.1629596 (cit. on pp. 7, 38, 47).

[74] V. Berstis. “Security and protection of data in the IBM System/38”. In: *Proceedings of the 7th Annual Symposium on Computer Architecture*. 1980. doi: 10.1145/800053.801932 (cit. on pp. 7, 74, 77, 78, 80).

[75] B. G. Utley. *IBM System-38: Technical Developments*. IBM Corp, 1980. isbn: 0933186037 (cit. on pp. 7, 77, 80).

[76] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. “IBM System/38 support for capability-based addressing”. In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. 1981 (cit. on pp. 7, 77, 80).

[77] M. Fillo et al. “The M-Machine multicomputer”. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 1995 (cit. on pp. 7, 22, 77, 78, 81, 138).

[78] D. Cox et al. “IBM AS/400 processor technology”. In: *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 1991. doi: 10.1109/ICCD.1991.139944 (cit. on pp. 7, 81).

[79] R. M. Norton. *Hardware support for compartmentalisation*. Tech. rep. UCAM-CL-TR-887. University of Cambridge, Computer Laboratory, 2016. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-887.html (cit. on pp. 7, 22, 76, 81).

[80] D. A. Patterson and C. H. Sequin. “RISC I: A Reduced Instruction Set VLSI Computer”. In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. 1981 (cit. on pp. 7, 22, 76).

[81] J. Woodruff et al. “CHERI Concentrate: Practical Compressed Capabilities”. In: *IEEE Transactions on Computers* (2019). doi: 10.1109/TC.2019.2914037 (cit. on pp. 7, 81, 87, 138, 141).

[82] R. N. Watson et al. “CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization”. In: *IEEE Symposium on Security and Privacy*. 2015. doi: 10.1109/SP.2015.9 (cit. on pp. 8, 24, 85, 93, 99–101).

[83] *Arm Morello System Development Platform Technical Reference Manual*. Tech. rep. Arm Limited, 2022. url: https://developer.arm.com/documentation/102278 (cit. on pp. 8, 25, 64, 75, 89, 91).

[84] Codasip. *CHERI security technology*. url: https://codasip.com/solutions/riscv-processor-safety-security/cheri/ (cit. on pp. 8, 75, 89–92, 154, 155).

[85] University of Cambridge. *The Digital Security by Design (DSbD) Initiative*. 2019. url: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/dsbd.html (cit. on pp. 8, 75).

[86] lowRISC. *Unveiling Sonata: Affordable CHERI Hardware for Embedded Systems*. 2024. url: https://lowrisc.org/news/2024/01/sonata-board-update/ (cit. on pp. 8, 64, 75, 91).

[87] S. Amar, T. Chen, D. Chisnall, et al. *CHERIoT: Rethinking Security for Low-Cost Embedded Systems*. Tech. rep. MSR-TR-2023-6. Microsoft, 2023. url: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/ (cit. on pp. 8, 25, 61, 64, 68, 73, 75, 86, 88, 89, 91, 98, 100, 159, 176).

[88] CHERI. *The CHERI LLVM Compiler Infrastructure*. url: https://github.com/CTSRD-CHERI/llvm-project (cit. on pp. 8, 93).

[89] B. 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*. 2019. doi: 10.1145/3297858.3304042 (cit. on pp. 8, 85, 94–96, 101).

[90] D. Lowther, D. Jacob, and J. Singer. “Morello MicroPython: A Python Interpreter for CHERI”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 2023. doi: 10.1145/3617651.3622991 (cit. on pp. 8, 101).

[91] B. Gutstein. *Memory Safety with CHERI Capabilities: Security Analysis, Language Interpreters, and Heap Temporal Safety*. Tech. rep. UCAM-CL-TR-975. University of Cambridge, Computer Laboratory, 2022. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-975.pdf (cit. on pp. 8, 101).

[92] D. Chisnall et al. “CHERI JNI: Sinking the Java Security Model into the C”. In: *SIGARCH Computer Architecture News* (2017). doi: 10.1145/3093337.3037725 (cit. on pp. 8, 101).

[93] P. Butcher, D. King, et al. *Security by Default - CHERI ISA Extensions Coupled with a Security-Enhanced Ada Runtime*. Tech. rep. AdaCore, 2024 (cit. on pp. 8, 101).

[94] S. Patel et al. “CHERI-Picking: Leveraging Capability Hardware for Prefetching”. In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. 2023. doi: 10.1145/3623759.3624553 (cit. on pp. 8, 101).

[95] R. N. Watson. *Department of Computer Science and Technology: CHERI-QEMU*. 2019. url: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-qemu.html (cit. on pp. 8, 102).

[96] Cambridge University. *CHERI-Spike*. url: https://github.com/CTSRD-CHERI/riscv-isa-sim (cit. on pp. 8, 102).

[97] Cambridge University. *CheriPlot*. url: https://github.com/CTSRD-CHERI/cheriplot (cit. on pp. 8, 102).

[98] CHERI Alliance / riscv-software-src. *OpenSBI with CHERI Support*. OpenSBI variant with CHERI register support for RISC-V. 2024–2025. url: https://github.com/cheri-alliance/opensbi (cit. on p. 8).

[99] Cambridge University. *CheriBSD*. url: https://github.com/CTSRD-CHERI/cheribsd (cit. on pp. 8, 61, 64, 96, 159).

[100] K. Wang, D. Kasatkin, et al. “Cherifying Linux: A Practical View on using CHERI”. In: *Proceedings of the 17th EuroSec*. 2024. doi: 10.1145/3642974.3652282 (cit. on pp. 8, 96, 159).

[101] Huawei. *Linux CHERI RISC-V project*. 2022. url: https://github.com/cheri-linux (cit. on pp. 8, 64, 75, 96).

[102] H. Almatary. “CHERI-seL4: Enhancing seL4’s C/C++ userspace memory safety using CHERI”. seL4 Summit 2023 presentation. 2024. url: https://sel4.systems//Foundation/Summit/2024/slides/cheri-morello.pdf (cit. on pp. 8, 98).

[103] J. Jackson. “CHERI-Zephyr”. In: *CHERITech*. 2024. url: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/cheritech24/02_01_Jennifer_Jackson_Minmin_Jiang.pdf (cit. on pp. 8, 64, 95, 97).

[104] A. M. Community. *Morello / Newlib*. 2020. url: https://git.morello-project.org/morello/newlib (cit. on pp. 8, 101).

[105] A. M. Community. *Morello / Musl libc*. 2021. url: https://git.morello-project.org/morello/musl-libc (cit. on pp. 8, 101).

[106] *CHERI Alliance - Industry-Led Security Technology*. 2024. url: https://cheri-alliance.org/ (cit. on pp. 8, 75).

[107] T. Aird, H. Almatary, J. Woodruff, et al. *RISC-V Specification for CHERI Extensions*. Tech. rep. RISC-V International, 2025. url: https://github.com/riscv/riscv-cheri/releases/download/v0.9.5/riscv-cheri.pdf (cit. on pp. 8, 25, 87, 90, 137, 153).

[108] L. Valente et al. “Shaheen: An Open, Secure, and Scalable RV64 SoC for Autonomous Nano-UAVs”. In: *2023 IEEE Hot Chips 35 Symposium (HCS)*. 2023. doi: 10.1109/HCS59251.2023.10254698 (cit. on pp. 11, 14, 69, 71).

[109] F. Marques et al. ““Interrupting” the Status Quo: A First Glance at the RISC-V Advanced Interrupt Architecture (AIA)”. In: *IEEE Access* (2024). doi: 10.1109/ACCESS.2024.3352114 (cit. on pp. 11, 14, 71).

[110] B. Sá et al. “Holistic RISC-V Virtualization: CVA6-based SoC”. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023. doi: 10.1145/3587135.3591436 (cit. on pp. 11, 14, 71).

[111] B. Sá, D. Ferraro, E. Matos, et al. *Advanced Security through Hardware Capabilities: A Comprehensive Survey on CHERI Technology*. Accepted at ACM Computing Surveys. 2026 (cit. on pp. 11, 71).

[112] B. Sá et al. “CVA6 RISC-V Virtualization: Architecture, Microarchitecture, and Design Space Exploration”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2023). doi: 10.1109/TVLSI.2023.3302837 (cit. on pp. 12, 71, 89, 140, 159).

[113] B. Sá, D. Ferraro, E. Matos, et al. *CVA6 RISC-V CHERI: Architecture, Microarchitecture, and Evaluation*. Submitted to IEEE Transactions on Computers. 2026 (cit. on pp. 12, 71).

[114] G. J. Popek and R. P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Communications of the ACM* (1974) (cit. on pp. 15, 114).

[115] P. Barham et al. “Xen and the art of virtualization”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. 2003. doi: 10.1145/945445.945462 (cit. on pp. 15, 17, 18, 107, 174).

[116] M. Rosenblum and T. Garfinkel. “Virtual machine monitors: current technology and future trends”. In: *Computer* (2005). doi: 10.1109/MC.2005.176 (cit. on p. 15).

[117] E. Bugnion et al. “Bringing Virtualization to the x86 Architecture with the Original VMware Workstation”. In: *ACM Transactions on Computer Systems* (2012). doi: 10.1145/2382553.2382554 (cit. on pp. 15–17).

[118] J. Martins et al. “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems”. In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. 2020. doi: 10.4230/OASIcs.NG-RES.2020.3 (cit. on pp. 15, 17, 18, 49, 60, 107, 111, 112, 140, 153, 158, 167).

[119] M. Cinque et al. “Virtualizing mixed-criticality systems: A survey on industrial trends and issues”. In: *Future Generation Computer Systems* (2022). doi: 10.1016/j.future.2021.12.002 (cit. on p. 15).

[120] Advanced Micro Devices. *Secure Virtual Machine Architecture Reference Manual*. Publication No. 33047, Rev. 3.01. 2005 (cit. on p. 16).

[121] Arm Ltd. *Arm Architecture Reference Manual for A-profile architecture*. 2025 (cit. on pp. 16, 42, 44, 45, 57).

[122] C. Dall and J. Nieh. “KVM/ARM: the design and implementation of the linux ARM hypervisor”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2014. doi: 10.1145/2541940.2541946 (cit. on pp. 16, 17).

[123] H. Li et al. “ACRN: A Big Little Hypervisor for IoT Development”. In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2019. doi: 10.1145/3313808.3313816 (cit. on p. 17).

[124] SYSGO GmbH. *PikeOS: Real-Time Operating System and Hypervisor*. 2025. url: https://www.sysgo.com/rtos-hypervisor (cit. on pp. 17, 48, 60).

[125] S. Stabellini. *True static partitioning with Xen Dom0-less*. Xen Project blog post describing Dom0-less static partitioning. 2019. url: https://xenproject.org/blog/true-static-partitioning-with-xen-dom0-less/ (cit. on pp. 17, 18, 49).

[126] B. QNX. *QNX Hypervisor for Safety - Product Brief*. 2019. url: https://aditech.in/wp-content/uploads/2020/03/QNX_HypervisorAutomotive_ProductBrief.pdf (cit. on pp. 17, 48).

[127] W. River. *Wind River Hypervisor*. 2010. url: https://www.cs.unc.edu/~anderson/teach/comp790/papers/Wind-River-Hypervisor.pdf (cit. on p. 17).

[128] F. Bellard. “QEMU, a fast and portable dynamic translator”. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. 2005 (cit. on pp. 17, 173).

[129] Oracle Corporation. *VirtualBox*. 2025. url: https://www.virtualbox.org (cit. on p. 17).

[130] eSOL. *eMCOS Hypervisor*. 2025. url: https://www.esol.com/embedded/product/emcos-hypervisor.html (cit. on p. 17).

[131] J. Martins and S. Pinto. "Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems". In: *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2023 (cit. on pp. 17, 18, 123, 148, 158, 167).

[132] S. AG. *Jailhouse: Linux-based Partitioning Hypervisor*. 2025. url: https://github.com/siemens/jailhouse (cit. on pp. 18, 49, 60).

[133] E. P. Gomes, G. De O. Aguiar, and G. Gracioli. "Implementation and Evaluation of MemGuard in the Bao Hypervisor". In: *XIV Brazilian Symposium on Computing Systems Engineering (SBESC)*. 2024. doi: 10.1109/SBESC65055.2024.10771909 (cit. on p. 19).

[134] A. Oliveira et al. "H-MBR: Hypervisor-Level Memory Bandwidth Reservation for Mixed Criticality Systems". In: *6th Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*. 2025. doi: 10.4230/OASIcs.NG-RES.2025.4 (cit. on pp. 19, 51, 52).

[135] D. Costa et al. "IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms". In: *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*. 2023. doi: 10.4230/OASIcs.NG-RES.2023.2 (cit. on pp. 19, 52).

[136] J. Peixoto et al. "BiRtIO: VirtIO for Real-Time Network Interface Sharing on the Bao Hypervisor". In: *IEEE Access* (2024). doi: 10.1109/ACCESS.2024.3512777 (cit. on pp. 19, 153, 174).

[137] R. Russell. "VirtIO: Towards a De-Facto Standard for Virtual I/O Devices". In: *SIGOPS Operating Systems Review* (2008). doi: 10.1145/1400097.1400108 (cit. on pp. 19, 60, 173).

[138] A. Santos et al. "Let's Get Physical: Rethinking the Static Partitioning Hypervisor Architecture for an MMU-Less Memory Model". In: *IEEE Access* (2025). doi: 10.1109/ACCESS.2025.3636061 (cit. on pp. 19, 49, 175).

[139] S. Pereira et al. "Bao-Enclave: Virtualization-based Enclaves for Arm". In: *IEEE 8th World Forum on Internet of Things (WF-IoT)*. 2022. doi: 10.1109/WF-IoT54382.2022.10152074 (cit. on pp. 19, 54).

[140] International Organization for Standardization and Society of Automotive Engineers. *ISO/SAE DIS 21434: Road vehicles - Cybersecurity engineering*. 2021. url: https://www.iso.org/standard/70918.html (cit. on p. 20).

[141] S. Pinto et al. "RISC-V Needs Secure 'Wheels': the MCU Initiator-Side Perspective". In: *CoRR* (2024). arXiv preprint (cit. on pp. 20, 49, 175).

[142] R. S. Fabry. "Capability-Based Addressing". In: *Communications of the ACM* (1974). doi: 10.1145/361011.361070 (cit. on pp. 20, 76, 77).

[143] J. H. Saltzer. “Protection and the Control of Information Sharing in Multics”. In: *Communications of the ACM* (1974). doi: 10.1145/361011.361067 (cit. on pp. 21, 23, 74, 138).

[144] B. Corporation. *The descriptor: A Definition of the B5000 Information Processing System*. Detroit, Mich., 1961 (cit. on pp. 21, 76).

[145] G. Klein et al. “Comprehensive formal verification of an OS microkernel”. In: *ACM Transactions on Computer Systems* (2014). doi: 10.1145/2560537 (cit. on p. 21).

[146] F. Pagano, L. Verderame, and A. Merlo. “Understanding Fuchsia Security”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* (2021). doi: 10.22667/JOWUA.2021.09.30.047 (cit. on pp. 21, 81).

[147] D. M. England. “Operating System of System 250”. In: 1972 (cit. on pp. 22, 77–80).

[148] D. M. England. “Capability Concept Mechanism and Structure in System 25”. In: 1974 (cit. on pp. 22, 77–80).

[149] I. Corporation. *iAPX 432 General Data Processor Architecture Reference Manual*. 1981 (cit. on pp. 22, 77–80).

[150] I. Corporation. *Introduction to the iAPX 432 Architecture*. 1981 (cit. on pp. 22, 77–80).

[151] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier, 1979 (cit. on pp. 22, 77, 79).

[152] R. M. Needham and R. D. Walker. “The Cambridge CAP Computer and Its Protection System”. In: *Proceedings of the 6th ACM Symposium on Operating Systems Principles*. 1977. doi: 10.1145/800214.806541 (cit. on pp. 22, 77–79, 138).

[153] J. M. Foster, I. F. Currie, et al. *Flex: A Working Computer with an Architecture Based on Procedure Values*. 1982 (cit. on pp. 22, 77, 78, 81).

[154] J. M. Foster, C. I. Moir, et al. *An Introduction to the FLEX Computer System*. 1979 (cit. on pp. 22, 77, 78).

[155] P. Vasek and K. Ghose. “A Fast Capability Extension to a RISC Architecture”. In: *Proceedings of EUROMICRO*. 1996. doi: 10.1109/EURMIC.1996.546488 (cit. on pp. 22, 77–81).

[156] P. Vasek and K. Ghose. “A Comparison of Two Context Allocation Approaches for Fast Protected Calls”. In: *Proceedings Fourth International Conference on High-Performance Computing*. 1997. doi: 10.1109/HIPC.1997.634463 (cit. on pp. 22, 77–81).

[157] U. Dhawan et al. “Hardware Support for Safety Interlocks and Introspection”. In: *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. 2012. doi: 10.1109/SASOW.2012.11 (cit. on pp. 22, 74, 77, 78, 80).

[158] L. Vilanova et al. “CODOMs: Protecting software with Code-centric memory Domains”. In: *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014. doi: 10.1109/ISCA.2014.6853202 (cit. on pp. 22, 77, 78, 80).

[159] G. Kane. *MIPS RISC Architecture*. Prentice Hall, 1988 (cit. on p. 22).

[160] D. L. Weaver and T. Germond, eds. *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1994 (cit. on p. 22).

[161] M. Miller, K. Yee, et al. *Capability Myths Demolished*. 2003 (cit. on pp. 22, 76, 136).

[162] R. N. M. Watson et al. “Capsicum: practical capabilities for UNIX”. In: *Proceedings of the 19th USENIX Conference on Security*. 2010 (cit. on p. 22).

[163] P. G. Neumann. “Fundamental Trustworthiness Principles in CHERI”. In: *New Solutions for Cybersecurity*. The MIT Press, 2018. doi: 10.7551/mitpress/11636.003.0009 (cit. on pp. 23, 74, 83).

[164] P. Rugg et al. “A Suite of Processors to Explore CHERI-RISC-V Micro Architecture”. In: *27th Euromicro Conference on Digital System Design (DSD)*. 2024. doi: 10.1109/DSD64264.2024.00054 (cit. on pp. 25, 75, 90–92, 155).

[165] Cambridge University. *CHERI Flute*. url: https://github.com/CTSRD-CHERI/Flute (cit. on pp. 25, 64, 87, 89, 90, 154, 155).

[166] Cambridge University. *CHERI Toooba*. url: https://github.com/CTSRD-CHERI/Toooba (cit. on pp. 25, 64, 87, 89, 90, 154, 155).

[167] A. Limited. *Arm Architecture Reference Manual Supplement - Morello for A-profile Architecture*. Tech. rep. Arm Limited, 2022. url: https://developer.arm.com/documentation/ddi0606/ (cit. on pp. 25, 86, 87, 91, 137).

[168] K. Asanović and D. A. Patterson. *Instruction Sets Should Be Free: The Case for RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, Univ. of California, Berkeley, 2014 (cit. on pp. 25, 111).

[169] Global Market Insights. *RISC-V Market Size, Share & Growth Report, 2025–2034*. Report ID: GMI14490. 2025. url: https://www.gminsights.com/industry-analysis/risc-v-market (cit. on p. 26).

[170] I. Editorial. *Omdia predicts RISC-V processors to reach 17 billion units by 2030*. Summarizes Omdia’s 17B-units-by-2030 forecast. 2024. url: https://www.iotinsider.com/news/omdia-predicts-risc-v-processors-will-grow-in-size/ (cit. on p. 26).

[171] e. Editorial. *Edge AI to help RISC-V take 25% of processor market by 2030*. Reports Omdia’s units/share outlook (incl. 17B figure). 2024. url: https://www.eenewseurope.com/en/edge-ai-to-help-risc-v-to-take-25-of-processor-market/ (cit. on p. 26).

[172] The SHD Group. *RISC-V Market Report: Application Forecasts in a Heterogeneous World*. 2024. url: https://theshdgroup.com/wp-content/uploads/2024/01/RISC-V-Market-Analysis-2024-Abridged-Report-2.pdf (cit. on p. 26).

[173] RISC-V International. *NVIDIA to Bring CUDA Platform Support to the RISC-V ISA*. 2025. url: https://riscv.org/ecosystem-news/2025/07/nvidia-to-bring-cuda-platform-support-to-the-risc-v/ (cit. on p. 26).

[174] Reuters. *China to Publish Policy to Boost RISC-V Chip Use Nationwide*. 2025. url: https://www.reuters.com/technology/china-publish-policy-boost-risc-v-chip-use-nationwide-sources-2025-03-04/ (cit. on p. 26).

[175] European Commission. *European Chips Act*. 2023. url: https://digital-strategy.ec.europa.eu/en/policies/european-chips-act (cit. on p. 26).

[176] A. Waterman and et al. "The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture, Document Version 20250925". In: *RISC-V Foundation, Jul* (2025) (cit. on pp. 26, 27).

[177] K. Asanović, A. Waterman, and P. Ashende. "The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 1.13". In: *RISC-V Foundation, Jul* (2025) (cit. on pp. 26, 27, 112, 114, 116, 138).

[178] OpenHW Group. *CVA6 RISC-V Core (CORE-V & OpenHW Group)*. 2025. url: https://github.com/openhwgroup/cva6 (cit. on p. 29).

[179] M. Cavalcante et al. "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020). doi: 10.1109/TVLSI.2019.2950087 (cit. on pp. 29, 116).

[180] S. Manoni et al. *CVA6-CFI: A First Glance at RISC-V Control-Flow Integrity Extensions*. 2026. arXiv: 2602.04991 [cs.AR]. url: https://arxiv.org/abs/2602.04991 (cit. on p. 29).

[181] R. Tedeschi et al. *CVA6S+: A Superscalar RISC-V Core with High-Throughput Memory Architecture*. 2025. arXiv: 2505.03762 [cs.AR] (cit. on p. 30).

[182] R. Tedeschi et al. *Culsans: An Efficient Snoop-based Coherency Unit for the CVA6 Open Source RISC-V Application Processor*. 2024. url: https://arxiv.org/abs/2407.19895 (cit. on p. 30).

[183] C. Fuguet. "HPDcache: Open-Source High-Performance L1 Data Cache for RISC-V Cores". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023. doi: 10.1145/3587135.3591413 (cit. on p. 30).

[184] A. Ottaviano et al. "Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2023). doi: 10.1109/TCSII.2023.3289186 (cit. on pp. 30, 66).

[185] P. Team. *Carfield*. url: https://pulp-platform.github.io/carfield/ (cit. on pp. 30, 66).

[186] G. Paulin, F. Zaruba, and S. Mach. *Occamy*. 2022. url: http://asic.ethz.ch/2022/Occamy.html (cit. on p. 30).

[187] P. Sauter et al. *Basilisk: A 34 mm2 End-to-End Open-Source 64-bit Linux-Capable RISC-V SoC in 130nm BiCMOS*. 2025. arXiv: 2505.10060 [cs.AR]. url: https://arxiv.org/abs/2505.10060 (cit. on p. 30).

[188] L. Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013. doi: 10.1109/SP.2013.13 (cit. on pp. 32–34, 36).

[189] M. Corporation. *CWE Top 25 Most Dangerous Software Weaknesses*. 2025. url: https://cwe.mitre.org/top25/archive/2025/2025\cwe\top25.html (cit. on p. 32).

[190] A. Sharma et al. "Rust for Embedded Systems: Current State and Open Problems". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2024. doi: 10.1145/3658644.3690275 (cit. on pp. 33, 35, 74, 136).

[191] M. C. E. Board. *CVE-2024-49775: Heap-based Buffer Overflow in Siemens User Management Component*. CVE Record, NVD. 2024. url: https://www.cve.org/CVERecord?id=CVE-2024-49775 (cit. on p. 33).

[192] M. C. E. Board. *CVE-2025-29828: Missing Release of Memory after Effective Lifetime in Windows Cryptographic Services*. CVE Record, NVD. 2025. url: https://www.cve.org/CVERecord?id=CVE-2025-29828 (cit. on p. 34).

[193] N. D. Matsakis and F. S. Klock. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. 2014. doi: 10.1145/2663171.2663188 (cit. on p. 35).

[194] SYSGO GmbH. *Rust Embedded on PikeOS*. 2024. url: https://www.sysgo.com/blog/article/rust-embedded-on-pikeos (cit. on p. 35).

[195] Bharath Subramanian. *Introducing Rusted Firmware-A (RF-A) - A Rust-Based reimagination of Trusted Firmware-A*. 2025. url: https://www.trustedfirmware.org/blog/rf-a-blog/ (cit. on p. 35).

[196] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 2011 (cit. on pp. 35, 48).

[197] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission (IEC), 2010 (cit. on pp. 35, 48).

[198] *Railway Applications - Communication, Signalling and Processing Systems*. CENELEC, 2011 (cit. on pp. 35, 48).

[199] S. Nagarakatte et al. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2009. doi: 10.1145/1542476.1542502 (cit. on pp. 36, 41).

[200] S. Nagarakatte et al. “CETS: Compiler-Enforced Temporal Safety for C”. In: *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*. 2010. doi: 10.1145/1806651.1806657 (cit. on pp. 36, 41).

[201] P. Akritidis et al. “Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors”. In: *Proceedings of the 18th USENIX Security Symposium (USENIX Security ’09)*. 2009 (cit. on p. 36).

[202] J. Zhang et al. “SanRazor: Reducing Redundant Sanitizer Checks in C/C++ Programs”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’21)*. Virtual Conference: USENIX Association, 2021, pp. 479–494. isbn: 978-1-939133-22-9 (cit. on p. 36).

[203] H. Ling et al. “GIANTSAN: Efficient Memory Sanitization with Segment Folding”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2024. doi: 10.1145/3620665.3640391 (cit. on p. 36).

[204] K. Cho et al. “RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust”. In: *33rd USENIX Security Symposium*. 2024 (cit. on p. 36).

[205] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996 (cit. on pp. 36, 57).

[206] E. D. Berger, B. G. Zorn, and K. S. McKinley. “Reconsidering Custom Memory Allocation”. In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2002. doi: 10.1145/582419.582421 (cit. on pp. 36, 57).

[207] A. M. Yang and T. Wrigstad. “Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK”. In: *ACM Transactions on Programming Languages and Systems* (2022) (cit. on pp. 36, 41).

[208] C. H. Flood et al. “An Open-Source Concurrent Compacting Garbage Collector for OpenJDK”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016. doi: 10.1145/2972206.2972210 (cit. on pp. 36, 41).

[209] G. Tene, B. Iyengar, and M. Wolf. “C4: Continuously Concurrent Compacting Collector”. In: *SIGPLAN Notices* (2011). doi: 10.1145/2076022.1993491 (cit. on p. 36).

[210] D. F. Bacon, P. Cheng, and V. T. Rajan. “A Real-Time Garbage Collector with Low Overhead and Consistent Utilization”. In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2003. doi: 10.1145/604131.604155 (cit. on p. 36).

[211] D. F. Bacon et al. “Syncopation: Generational Real-time Garbage Collection in the Metronome”. In: *SIGPLAN Notices* (2005). doi: 10.1145/1070891.1065937 (cit. on p. 36).

[212] H.-J. Boehm and M. Weiser. “Garbage Collection in an Uncooperative Environment”. In: *Softw. Pract. Exper.* (1988). doi: 10.1002/spe.4380180902 (cit. on pp. 36, 41).

[213] B. Organization. *The Boehm–Demers–Weiser Conservative Garbage Collector*. 1988. url: https://www.hboehm.info/gc/ (cit. on p. 36).

[214] H.-J. Boehm and M. Spertus. “Garbage Collection in the Next C++ Standard”. In: *Proceedings of the 2009 International Symposium on Memory Management*. 2009. doi: 10.1145/1542431.1542437 (cit. on p. 36).

[215] M. Schoeberl and J. Vitek. “Garbage collection for safety critical Java”. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*. 2007. doi: 10.1145/1288940.1288953 (cit. on p. 36).

[216] T. Würthinger, C. Wimmer, and H. Mössenböck. “Array Bounds Check Elimination for the Java HotSpot™ Client Compiler”. In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. 2007. doi: 10.1145/1294325.1294343 (cit. on p. 36).

[217] F. Qian, L. J. Hendren, and C. Verbrugge. “A Comprehensive Approach to Array Bounds Check Elimination for Java”. In: *Proceedings of the 11th International Conference on Compiler Construction*. 2002 (cit. on p. 36).

[218] J. Devietti et al. “Hardbound: Architectural Support for Spatial Safety of the C Programming Language”. In: *Proceedings of the 13th International Conference on ASPLOS*. 2008. doi: 10.1145/1346281.1346295 (cit. on pp. 37, 41).

[219] E. Witchel, J. Cates, and K. Asanović. “Mondrian memory protection”. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2002. doi: 10.1145/605397.605429 (cit. on p. 37).

[220] Arm. *Armv8.5-A Memory Tagging Extension*. 2021. url: https://developer.arm.com/documentation/102925/latest/ (cit. on pp. 37, 57, 74).

[221] RISC-V J Extension Task Group. *RISC-V Pointer Masking Extension*. Tech. rep. Ratified Version 1.0. RISC-V International, 2024 (cit. on p. 37).

[222] R. M. farkhani, M. Ahmadi, and L. Lu. “PTAuth: Temporal Memory Safety via Robust Points-to Authentication”. In: *Proceedings of the 30th USENIX Security Symposium*. 2021 (cit. on pp. 37, 41).

[223] M. Unterguggenberger et al. “Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging”. In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 2023. doi: 10.1145/3579856.3590331 (cit. on p. 38).

[224] L. Lamster et al. “HashTag: Hash-based Integrity Protection for Tagged Architectures”. In: *32nd USENIX Security Symposium*. 2023 (cit. on p. 38).

[225] Y. Kim, J. Lee, and H. Kim. “Hardware-Based Always-On Heap Memory Safety”. In: *53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2020 (cit. on pp. 38, 57).

[226] Q. T. Inc. *Pointer Authentication on ARMv8.3*. 2017 (cit. on pp. 38, 57).

[227] X. Leroy. “A Formally Verified Compiler Back-end”. In: *J. Autom. Reason.* (2009). doi: 10.1007/s10817-009-9155-4 (cit. on pp. 38, 41).

[228] J. Kang et al. “Crellvm: Verified Credible Compilation for LLVM”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018. doi: 10.1145/3192366.3192377 (cit. on p. 38).

[229] J. Zhao et al. “Formalizing the LLVM Intermediate Representation for Verified Program Transformations”. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012. doi: 10.1145/2103656.2103709 (cit. on p. 38).

[230] B. Jacobs et al. “VeriFast: a Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *Proceedings of the Third International Conference on NASA Formal Methods*. 2011 (cit. on pp. 38, 41, 57).

[231] E. Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 2009 (cit. on pp. 38, 57).

[232] D. Gao and T. Melham. “End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers”. In: *2021 Formal Methods in Computer Aided Design (FMCAD)*. 2021. doi: 10.34727/2021/isbn.978-3-85448-046-4_10 (cit. on p. 38).

[233] A. Armstrong, T. Bauereiss, B. Campbell, et al. “ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS”. In: *Proceedings of the ACM on Programming Languages* (2019). doi: 10.1145/3290384 (cit. on pp. 38, 102).

[234] R. Gu et al. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels”. In: *12th USENIX Symposium on Operating Systems Design and Implementation*. 2016 (cit. on p. 38).

[235] P. Cousot, R. Giacobazzi, and F. Ranzato. “A²I: abstract² interpretation”. In: *Proceedings of the ACM on Programming Languages* (2019). doi: 10.1145/3290355 (cit. on p. 39).

[236] C. Calcagno et al. “Moving Fast with Software Verification”. In: *NASA Formal Methods Symposium*. 2015 (cit. on pp. 39, 41).

[237] T. C. Team. *Clang Static Analyzer*. Accessed on: 15-02-2026. url: https://clang.llvm.org/docs/ClangStaticAnalyzer.html (cit. on pp. 39, 41, 57).

[238] P. J. Guo and D. Engler. “Linux Kernel Developer Responses to Static Analysis Bug Reports”. In: *USENIX Annual Technical Conference*. 2009 (cit. on p. 39).

[239] N. Nethercote and J. Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007. doi: 10.1145/1250734.1250746 (cit. on pp. 39, 41, 57).

[240] D. Bruening and Q. Zhao. “Practical Memory Checking with Dr. Memory”. In: *International Symposium on Code Generation and Optimization*. 2011. doi: 10.1109/CGO.2011.5764689 (cit. on pp. 39, 41).

[241] B. P. Miller, L. Fredriksen, and B. So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Communications of the ACM* (1990). doi: 10.1145/96267.96279 (cit. on p. 39).

[242] A. Fioraldi et al. *Dissecting American Fuzzy Lop: A FuzzBench Evaluation*. 2023. doi: 10.1145/3580596 (cit. on pp. 39, 57).

[243] M. Böhme, V.-T. Pham, and A. Roychoudhury. “Coverage-based Greybox Fuzzing as Markov Chain”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016 (cit. on pp. 39, 57).

[244] A. Fioraldi et al. “AFL++: Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT)*. 2020 (cit. on pp. 39, 41, 57).

[245] P. Chen and H. Chen. “Angora: Efficient Fuzzing by Principled Search”. In: *IEEE Symposium on S&P*. 2018 (cit. on p. 39).

[246] R. Swiecki. *Honggfuzz*. 2018. url: https://github.com/google/honggfuzz (cit. on p. 39).

[247] K. Serebryany. “Continuous Fuzzing with libFuzzer and AddressSanitizer”. In: *IEEE Cybersecurity Development (SecDev)*. 2016 (cit. on pp. 39, 41).

[248] C. Lemieux and K. Sen. “FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage”. In: *ACM/IEEE International Conference on Automated Software Engineering*. 2018 (cit. on p. 39).

[249] C. Lyu et al. “MOPT: Optimized Mutation Scheduling for Fuzzers”. In: *28th USENIX Security Symposium*. 2019 (cit. on p. 39).

[250] S. Poeplau and A. Francillon. “Symbolic Execution with SymCC: Don’t Interpret, Compile!” In: *29th USENIX Security Symposium*. 2020 (cit. on p. 39).

[251] A. Fioraldi, D. C. D’Elia, and L. Querzoni. “Fuzzing Binaries for Memory Safety Errors with QASan”. In: *IEEE Secure Development (SecDev)*. 2020 (cit. on pp. 40, 41).

[252] M. Marini et al. “QMSan: Efficiently Detecting Uninitialized Memory Errors During Fuzzing”. In: *Network and Distributed System Security Symposium*. 2025 (cit. on pp. 40, 41).

[253] S. Österlund et al. “ParmeSan: Sanitizer-guided Greybox Fuzzing”. In: *29th USENIX Security Symposium*. 2020 (cit. on p. 40).

[254] N. Coppik, O. Schwahn, and N. Suri. “MemFuzz: Using Memory Accesses to Guide Fuzzing”. In: *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019. doi: 10.1109/ICST.2019.00015 (cit. on p. 40).

[255] M. Rostami et al. *Fuzzerfly Effect: Hardware Fuzzing for Memory Safety*. 2024. doi: 10.1109/MSEC.2024.3365070 (cit. on p. 40).

[256] S. Schumilo et al. “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels”. In: *26th USENIX Security Symposium*. 2017 (cit. on p. 40).

[257] C. Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *7th USENIX Security Symposium*. 1998 (cit. on pp. 42, 45, 57).

[258] H. Etoh. *ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks*. 2003 (cit. on pp. 42, 45, 57).

[259] K.-S. Lhee and S. J. Chapin. “Buffer Overflow and Format String Overflow Vulnerabilities”. In: *Softw. Pract. Exper.* (2003). doi: 10.1002/spe.515 (cit. on pp. 42, 57).

[260] T. C. Team. *SafeStack*. url: https://clang.llvm.org/docs/SafeStack.html (cit. on pp. 42, 45, 57).

[261] T. C. Team. *ShadowCallStack*. url: https://clang.llvm.org/docs/ShadowCallStack.html (cit. on pp. 42, 45, 57).

[262] V. Shanbhogue, D. Gupta, and R. Sahita. “Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity”. In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2019. doi: 10.1145/3337167.3337175 (cit. on pp. 42, 44, 45, 57).

[263] H. Liljestrand et al. “PAC it up: Towards Pointer Integrity using ARM Pointer Authentication”. In: *28th USENIX Security Symposium*. 2019 (cit. on p. 42).

[264] H. Liljestrand et al. “Protecting the Stack with PACed Canaries”. In: *Proceedings of the 4th Workshop on System Software for Trusted Execution*. 2019. doi: 10.1145/3342559.3365336 (cit. on p. 42).

[265] H. Liljestrand et al. “PACStack: an Authenticated Call Stack”. In: *30th USENIX Security Symposium*. 2021 (cit. on p. 42).

[266] E. D. Berger and B. G. Zorn. “DieHard: Probabilistic Memory Safety for Unsafe Languages”. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2006. doi: 10.1145/1133981.1134000 (cit. on pp. 42, 45, 58).

[267] G. Novark, E. D. Berger, and B. G. Zorn. “Exterminator: Automatically Correcting Memory Errors with High Probability”. In: *SIGPLAN Notices* (2007). doi: 10.1145/1273442.1250736 (cit. on pp. 42, 58).

[268] G. Novark and E. D. Berger. “DieHarder: Securing the Heap”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. 2010. doi: 10.1145/1866307.1866371 (cit. on pp. 42, 45, 58).

[269] B. Liu, P. Olivier, and B. Ravindran. “SlimGuard: A Secure and Memory-Efficient Heap Allocator”. In: *Proceedings of the 20th International Middleware Conference*. 2019. doi: 10.1145/3361525.3361532 (cit. on pp. 42, 58).

[270] S. Silvestro et al. “FreeGuard: A Faster Secure Heap Allocator”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017. doi: 10.1145/3133956.3133957 (cit. on pp. 42, 45).

[271] S. Silvestro et al. “Guarder: A Tunable Secure Allocator”. In: *27th USENIX Security Symposium*. 2018 (cit. on pp. 42, 45).

[272] R. Wang, M. Xu, and N. Asokan. “SeMalloc: Semantics-Informed Memory Allocator”. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2024. doi: 10.1145/3658644.3670363 (cit. on pp. 42, 45).

[273] R. Wang, M. Xu, and N. Asokan. “S2malloc: Statistically Secure Allocator for Use-After-Free Protection and More”. In: *Proceedings on the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2024. doi: 10.1007/978-3-031-64171-8_2 (cit. on pp. 42, 45).

[274] Z. Lin et al. “CAMP: Compiler and Allocator-based Heap Memory Protection”. In: *USENIX Security Symposium*. 2024 (cit. on pp. 42, 45).

[275] K. Huang et al. “Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2024. doi: 10.1145/3658644.3690310 (cit. on pp. 42, 45).

[276] P. Mao et al. “Exploiting Android’s Hardened Memory Allocator”. In: *18th USENIX WOOT*. 2024 (cit. on p. 42).

[277] S. Bhatkar, D. C. DuVarney, and R. Sekar. “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits”. In: *12th USENIX Security Symposium*. 2003 (cit. on pp. 43, 45, 58).

[278] H. Shacham et al. “On the effectiveness of address-space randomization”. In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. 2004 (cit. on p. 43).

[279] A. R. Nurmukhametov et al. “Fine-Grained Address Space Layout Randomization on Program Load”. In: *Programming and Computing Software* (2018) (cit. on pp. 43, 45).

[280] M. Backes and S. Nürnberger. “Oxymoron: making fine-grained memory randomization practical by allowing code sharing”. In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. 2014 (cit. on pp. 43, 45).

[281] A. Bittau et al. “Hacking Blind”. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 2014. doi: 10.1109/SP.2014.22 (cit. on pp. 43, 45).

[282] R. Hund, C. Willems, and T. Holz. “Practical Timing Side Channel Attacks Against Kernel Space ASLR”. In: *IEEE Symposium on S&P*. 2013. doi: 10.1109/SP.2013.23 (cit. on pp. 43, 45).

[283] L. Binosi et al. “The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2024. doi: 10.1145/3658644.3690239 (cit. on p. 43).

[284] D. Gruss et al. “KASLR is Dead: Long Live KASLR”. In: *9th International Symposium on Engineering Secure Software and Systems (ESSoS)*. 2017 (cit. on pp. 43, 45).

[285] B. Gras et al. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *Network and Distributed System Security Symposium (NDSS)*. 2017 (cit. on pp. 43, 45).

[286] R. Roemer et al. “Return-Oriented Programming: Systems, Languages, and Applications”. In: *ACM Transactions on Information and System Security* (2012). doi: 10.1145/2133375.2133377 (cit. on pp. 43, 45).

[287] Y. Chen et al. “NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64”. In: *IEEE Symposium on S&P*. 2017. doi: 10.1109/SP.2017.30 (cit. on pp. 43, 45, 58).

[288] D. Kwon et al. “uXOM: Efficient eXecute-Only Memory on ARM Cortex-M”. In: *USENIX Security Symposium*. 2019 (cit. on pp. 43, 45, 58).

[289] M. Schink and J. Obermaier. “Taking a Look into Execute-Only Memory”. In: *13th USENIX WOOT*. 2019 (cit. on pp. 43, 45).

[290] N. Burow et al. “Control-Flow Integrity: Precision, Security, and Performance”. In: *ACM Computing Surveys* (2017). doi: 10.1145/3054924 (cit. on pp. 43, 58).

[291] M. Abadi et al. “Control-Flow Integrity”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. 2005. doi: 10.1145/1102120.1102165 (cit. on pp. 43, 45).

[292] M. Abadi et al. “Control-Flow Integrity Principles, Implementations, and Applications”. In: *ACM Transactions on Information and System Security* (2009) (cit. on pp. 43, 45).

[293] C. Zhang et al. “Practical Control Flow Integrity and Randomization for Binary Executables”. In: *IEEE S&P*. 2013 (cit. on p. 43).

[294] LLVM Project. *Control Flow Integrity: Design Documentation*. url: https://clang.llvm.org/docs/ControlFlowIntegrity.html (cit. on pp. 43, 45).

[295] B. Niu and G. Tan. “Per-Input Control-Flow Integrity”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015 (cit. on p. 44).

[296] V. van der Veen et al. “A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level”. In: *IEEE Symposium on S&P*. 2016. doi: 10.1109/SP.2016.60 (cit. on pp. 44, 45).

[297] C. Zhang et al. “VTrust: Regaining Trust on Virtual Calls”. In: *NDSS*. 2016 (cit. on pp. 44, 45).

[298] T. H. Dang, P. Maniatis, and D. Wagner. “The Performance Cost of Shadow Stacks and Stack Canaries”. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 2015. doi: 10.1145/2714576.2714635 (cit. on p. 44).

[299] V. Kuznetsov et al. “Code-Pointer Integrity”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014 (cit. on pp. 44, 45, 58).

[300] I. Qualcomm Technologies. *Pointer Authentication on ARMv8.3*. 2017. url: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf (cit. on pp. 44, 45).

[301] Y. Kim et al. “Hardware-Assisted Code-Pointer Tagging for Forward-Edge Control-Flow Integrity”. In: *IEEE Computer Architecture Letters* (2023). doi: 10.1109/LCA.2023.3306326 (cit. on pp. 44, 45).

[302] M. Castro, M. Costa, and T. Harris. “Securing Software by Enforcing Data-flow Integrity”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006 (cit. on pp. 44, 45, 58).

[303] P. Akritidis et al. “Preventing Memory Error Exploits with WIT”. In: *IEEE S&P*. 2008 (cit. on pp. 44, 45).

[304] L. Cheng et al. “Exploitation Techniques and Defenses for Data-Oriented Attacks”. In: *IEEE Cybersecurity Development (SecDev)*. 2019. doi: 10.1109/SecDev.2019.00022 (cit. on pp. 44, 45).

[305] L. Feng et al. “Toward Taming the Overhead Monster for Data-flow Integrity”. In: *ACM Transactions Design Automation of Electronic Systems* (2021). doi: 10.1145/3490176 (cit. on pp. 44, 45).

[306] J. Liedtke. “On Micro-Kernel Construction”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. 1995. doi: 10.1145/224056.224075 (cit. on pp. 46, 60).

[307] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. 9th. Wiley Publishing, 2012. isbn: 1118063333 (cit. on p. 46).

[308] G. Heiser and B. Leslie. “The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors”. In: *Proceedings of the 1st ACM Asia-Pacific Workshop on Workshop on Systems*. 2010. doi: 10.1145/1851276.1851282 (cit. on p. 47).

[309] Q. Wang et al. “SPeCK: a kernel for scalable predictability”. In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 2015. doi: 10.1109/RTAS.2015.7108434 (cit. on pp. 47, 48).

[310] I. Kuz et al. “CAmkES: A component model for secure microkernel-based embedded systems”. In: *Journal of Systems and Software* (2007). doi: 10.1016/j.jss.2006.08.039 (cit. on p. 48).

[311] M. Paturel, I. Subasinghe, and G. Heiser. “First steps in verifying the seL4 Core Platform”. In: *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 2023. doi: 10.1145/3609510.3609821 (cit. on p. 48).

[312] J. M. Rushby. “Design and Verification of Secure Systems”. In: *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*. 1981. doi: 10.1145/800216.806586 (cit. on pp. 48, 60).

[313] Wind River Systems. *Helix Virtualization Platform*. Commercial RTOS-based virtualization platform for safety-critical embedded systems. 2023. url: https://www.windriver.com/products/helix-virtualization-platform (cit. on p. 48).

[314] Green Hills Software. *INTEGRITY Multivisor: Secure Virtualization for Embedded Systems*. Accessed on: 06-02-2026. 2024. url: https://www.ghs.com/products/rtos/integrity_virtualization.html (cit. on p. 48).

[315] Lynx Software Technologies. *LynxSecure Separation Kernel Hypervisor*. url: https://www.lynx.com/products/lynxsecure-separation-kernel-hypervisor (cit. on pp. 48, 60).

[316] Wind River Systems. *VxWorks Real-Time Operating System*. url: https://www.windriver.com/products/vxworks (cit. on p. 49).

[317] Green Hills Software. *INTEGRITY RTOS*. url: https://www.ghs.com/products/rtos/integrity.html (cit. on p. 49).

[318] BlackBerry QNX. *QNX Neutrino RTOS*. url: https://www.qnx.com/products/intl/neutrino_rtos/ (cit. on p. 49).

[319] Lynx Software Technologies. *LynxSecure*. url: https://www.lynx.com/products/lynxos-178-do-178c-certified-posix-rtos (cit. on p. 49).

[320] U. Steinberg and B. Kauer. “NOVA: a microhypervisor-based secure virtualization architecture”. In: *Proceedings of the 5th European Conference on Computer Systems*. 2010. doi: 10.1145/1755913.1755935 (cit. on p. 49).

[321] R. West et al. “A Virtualized Separation Kernel for Mixed-Criticality Systems”. In: *ACM Transactions on Computer Systems* (2016). doi: 10.1145/2935748 (cit. on p. 49).

[322] S. Pinto et al. “LTZVisor: TrustZone is the Key”. In: *29th Euromicro Conference on Real-Time Systems (ECRTS)*. 2017. doi: 10.4230/LIPIcs.ECRTS.2017.4 (cit. on p. 49).

[323] J. Martins et al. “uRTZVisor: A Secure and Safe Real-Time Hypervisor”. In: *Electronics* (2017). doi: 10.3390/electronics6040093 (cit. on p. 49).

[324] S. Pinto et al. “Lightweight multicore virtualization architecture exploiting ARM TrustZone”. In: *43rd Annual Conference of the IEEE Industrial Electronics Society*. 2017. doi: 10.1109/IECON.2017.8216603 (cit. on p. 49).

[325] G. Hu, Z. He, and R. B. Lee. “SoK: Hardware Defenses Against Speculative Execution Attacks”. In: *International Symposium on Secure and Private Execution Environment Design (SEED)*. 2021. doi: 10.1109/SEED51797.2021.00023 (cit. on p. 50).

[326] P. Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *IEEE Symposium on Security and Privacy (SP)*. 2019. doi: 10.1109/SP.2019.00002 (cit. on pp. 50, 108).

[327] X. Zhang, S. Dwarkadas, and K. Shen. “Towards Practical Page Coloring-Based Multicore Cache Management”. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. 2009. doi: 10.1145/1519065.1519076 (cit. on pp. 50, 51, 61).

[328] F. Liu et al. “CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016. doi: 10.1109/HPCA.2016.7446082 (cit. on pp. 50, 51).

[329] D. Townley et al. “Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks”. In: *31st USENIX Security Symposium*. 2022 (cit. on pp. 50, 51).

[330] V. Kiriansky et al. “DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors”. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018. doi: 10.1109/MICRO.2018.00083 (cit. on pp. 50, 51).

[331] M. K. Qureshi. “CEASER: Mitigating Conflict-Based Cache Attacks Via Encrypted-Address and Remapping”. In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 2018. doi: 10.1109/MICRO.2018.00068 (cit. on pp. 50, 51).

[332] J. Herter et al. “CAMA: A Predictable Cache-Aware Memory Allocator”. In: *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*. 2011. doi: 10.1109/ECRTS.2011.11 (cit. on p. 51).

[333] C. Sullivan et al. “Per-Bank Bandwidth Regulation of Shared Last-Level Cache for Real-Time Systems”. In: *IEEE RTSS*. 2024. doi: 10.1109/RTSS62706.2024.00036 (cit. on p. 51).

[334] M. Masmano et al. “TLSF: A New Dynamic Memory Allocator for Real-Time Systems”. In: *Proceedings. 16th Euromicro Conference on Real-Time Systems*. 2004. doi: 10.1109/EMRTS.2004.1311009 (cit. on p. 51).

[335] B. C. Ward et al. “Making Shared Caches More Predictable on Multicore Platforms”. In: *25th Euromicro Conference on Real-Time Systems*. 2013. doi: 10.1109/ECRTS.2013.26 (cit. on p. 51).

[336] S. A. Panchamukhi and F. Mueller. “Providing task isolation via TLB coloring”. In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 2015. doi: 10.1109/RTAS.2015.7108391 (cit. on p. 51).

[337] S. Deng, W. Xiong, and J. Szefer. “Secure TLBs”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019. doi: 10.1145/3307650.3322238 (cit. on pp. 51, 61).

[338] S. Groesbrink and T. Kerstan. “Modular Paging with Dynamic TLB Partitioning for Embedded Real-Time Systems”. In: *International Symposium on Industrial Embedded Systems*. 2008. doi: 10.1109/SIES.2008.4577713 (cit. on pp. 51, 61).

[339] F. Stolz et al. *Risky Translations: Securing TLBs against Timing Side Channels*. Cryptology ePrint Archive, Paper 2022/1394. 2022 (cit. on p. 51).

[340] Intel Corporation. *Speculative Execution Side Channel Mitigations*. Version 2.0. 2021 (cit. on pp. 51, 52, 61).

[341] L. Zhao et al. “HyBP: Hybrid Isolation-Randomization Secure Branch Predictor”. In: *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022. doi: 10.1109/HPCA53966.2022.00033 (cit. on pp. 51, 52).

[342] J. Lee, Y. Ishii, and D. Sunwoo. “Securing Branch Predictors with Two-Level Encryption”. In: *ACM Transactions Architecture and Code Optimization* (2020). doi: 10.1145/3404189 (cit. on pp. 51, 52).

[343] T. Zhang et al. “STBPU: A Reasonably Secure Branch Prediction Unit”. In: *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2022. doi: 10.1109/DSN53405.2022.00023 (cit. on pp. 51, 52).

[344] N. Wistoff et al. “Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning”. In: *IEEE Transactions on Computers* (2023). doi: 10.1109/TC.2022.3212636 (cit. on pp. 51, 52).

[345] H. Yun et al. “MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-Core Platforms”. In: *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013. doi: 10.1109/RTAS.2013.6531079 (cit. on pp. 51, 52, 61).

[346] E. P. Gomes et al. “Memguard-RW: Improved Real-Time Memory Bandwidth Regulation Within a Hypervisor”. In: *28th International Symposium on Real-Time Distributed Computing (ISORC)*. 2025. doi: 10.1109/ISORC65339.2025.00021 (cit. on pp. 51, 52, 61).

[347] F. Farshchi, Q. Huang, and H. Yun. “BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020. doi: 10.1109/RTAS48715.2020.00011 (cit. on pp. 51, 52, 61).

[348] A. Zuepke et al. “MemPol: Polling-Based Microsecond-Scale Per-Core Memory Bandwidth Regulation”. In: *Real-Time Systems* (2024). doi: 10.1007/s11241-024-09422-8 (cit. on pp. 51, 52, 61).

[349] T. Benz et al. “AXI-REALM: Safe, Modular and Lightweight Traffic Monitoring and Regulation for Heterogeneous Mixed-Criticality Systems”. In: *IEEE Transactions on Computers* (2025). doi: 10.1109/TC.2025.3584530 (cit. on pp. 51, 52, 61).

[350] M. Hassan, H. Patel, and R. Pellizzoni. “PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems”. In: *ACM Transactions Embedded Computing Systems* (2017). doi: 10.1145/3019611 (cit. on pp. 51, 52).

[351] S. Altmeyer et al. “Evaluation of Cache Partitioning for Hard Real-Time Systems”. In: *26th Euromicro Conference on Real-Time Systems*. 2014. doi: 10.1109/ECRTS.2014.11 (cit. on p. 50).

[352] J. Yu et al. “Synchronization Storage Channels (S2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions”. In: *32nd USENIX Security Symposium*. 2023 (cit. on p. 50).

[353] G. Gracioli et al. “A Survey on Cache Management Mechanisms for Real-Time Embedded Systems”. In: *ACM Computing Surveys* (2015). doi: 10.1145/2830555 (cit. on pp. 50, 61).

[354] M. Chisholm et al. “Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2016. doi: 10.1109/RTSS.2016.015 (cit. on p. 50).

[355] X. Wang et al. “Exclusive Hierarchies for Predictable Sharing in Last-Level Cache”. In: *IEEE 30th RTAS*. 2024. doi: 10.1109/RTAS61025.2024.00023 (cit. on p. 50).

[356] B. Gras et al. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *27th USENIX Security Symposium*. 2018 (cit. on p. 51).

[357] A. Tatar et al. “TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering”. In: *31st USENIX Security Symposium*. 2022 (cit. on p. 51).

[358] J. Kim, H. Jang, and Y. Shin. “A Survey of Side-Channel Attacks on Branch Prediction Units”. In: *ACM Computing Surveys* (2025). doi: 10.1145/3734218 (cit. on pp. 51, 61).

[359] I. Vougioukas et al. “BRB: Mitigating Branch Predictor Side-Channels”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019. doi: 10.1109/HPCA.2019.00058 (cit. on p. 52).

[360] L. Zhao et al. “A Lightweight Isolation Mechanism for Secure Branch Predictors”. In: *58th IEEE Design Automation Conference (DAC)*. 2021. doi: 10.1109/DAC18074.2021.9586178 (cit. on p. 52).

[361] H. Yavarzadeh et al. “Half&Half: Demystifying Intel’s Directional Branch Predictors for Fast, Secure Partitioned Execution”. In: *IEEE Symposium on S&P*. 2023. doi: 10.1109/SP46215.2023.10179415 (cit. on p. 52).

[362] P. Pessl et al. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *25th USENIX Security Symposium*. 2016 (cit. on p. 52).

[363] C. Rodrigues, D. Oliveira, and S. Pinto. “BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect”. In: *IEEE Symposium on Security and Privacy (SP)*. 2024. doi: 10.1109/SP54263.2024.00062 (cit. on p. 52).

[364] D. Costa, J. Martins, and S. Pinto. “Beyond the Bermuda Triangle of Contention: IOMMU Interference in Mixed Criticality Systems”. In: *IEEE 31st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2025. doi: 10.1109/RTCSA66114.2025.00027 (cit. on p. 52).

[365] A. Savino et al. “Multicore DRAM Bank-& Row-Conflict Bomb for Timing Attacks in Mixed-Criticality Systems ”. In: *IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*. 2024. doi: 10.1109/ISORC61049.2024.10551337 (cit. on p. 52).

[366] H. Yun et al. “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms”. In: *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014. doi: 10.1109/RTAS.2014.6925999 (cit. on p. 52).

[367] D. Cerdeira et al. “AnyTEE: An Open and Interoperable Software Defined TEE Framework”. In: *IEEE Access* (2025). doi: 10.1109/ACCESS.2025.3581487 (cit. on pp. 53, 54).

[368] V. Costan and S. Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016). url: https://api.semanticscholar.org/CorpusID:28642809 (cit. on pp. 53, 54, 60).

[369] Linaro. *OP-TEE: Open Portable Trusted Execution Environment*. Accessed on: 19-02-2026. url: https://www.trustedfirmware.org/projects/op-tee/ (cit. on p. 53).

[370] GlobalPlatform Tecnology. *TEE System Architecture*. Tech. rep. Version 1.3, Ref: GPD_SPE_009. GlobalPlatform, 2022 (cit. on p. 53).

[371] Arm Limited. *Arm Firmware Framework for Arm (FF-A) A-profile*. Tech. rep. DEN0077. Version 1.5. Arm Ltd., 2026 (cit. on p. 53).

[372] J. Wang et al. “RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone”. In: *IEEE Symposium on S&P*. 2022. doi: 10.1109/SP46214.2022.9833604 (cit. on p. 53).

[373] T. Van Eyck et al. “Mr-TEE: Practical Trusted Execution of Mixed-Criticality Code”. In: *Proceedings of the 24th International Middleware Conference: Industrial Track*. Middleware ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 22–28. isbn: 9798400704277. doi: 10.1145/3626562.3626831. url: https://doi.org/10.1145/3626562.3626831 (cit. on p. 54).

[374] S. Pinto et al. “FreeTEE: When Real-Time and Security Meet”. In: *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. 2015. doi: 10.1109/ETFA.2015.7301571 (cit. on p. 54).

[375] D. Oliveira, T. Gomes, and S. Pinto. “uTango: An Open-Source TEE for IoT Devices”. In: *IEEE Access* (2022). doi: 10.1109/ACCESS.2022.3152781 (cit. on p. 54).

[376] D. Kaplan, J. Powell, and T. Woller. *White Paper: AMD Memory Encryption*. Tech. rep. AMD, 2021 (cit. on pp. 54, 60, 108).

[377] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium*. 2016 (cit. on pp. 54, 60).

[378] S. Weiser et al. “TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V”. In: *Proceedings of the NDSS*. 2019. doi: 10.14722/ndss.2019.23068 (cit. on p. 54).

[379] D. Schrammel et al. “SPEAR-V: Secure and Practical Enclave Architecture for RISC-V”. In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 2023. doi: 10.1145/3579856.3595784 (cit. on p. 54).

[380] J. Noorman et al. “Sancus: Low-Cost Trustworthy Extensible Networked Devices with a Zero-Software Trusted Computing Base”. In: *Proceedings of the 22nd USENIX Conference on Security*. 2013 (cit. on p. 54).

[381] J. Noorman et al. “Sancus 2.0: A Low-Cost Security Architecture for IoT Devices”. In: *ACM Transactions on Privacy and Security* (2017). doi: 10.1145/3079763 (cit. on p. 54).

[382] R. Bahmani et al. “CURE: A Security Architecture with CUstomizable and Resilient Enclaves”. In: *30th USENIX Security Symposium*. 2021 (cit. on pp. 54, 108).

[383] F. Alder et al. “Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021. doi: 10.1145/3460120.3484782 (cit. on p. 54).

[384] M. Kuhne, S. Volos, and S. Shinde. “DORAMI: Privilege Separating Security Monitor on RISC-V TEEs”. In: *Proceedings of the 34th USENIX Conference on Security Symposium*. 2025 (cit. on p. 54).

[385] Y. Jia et al. “HyperEnclave: An Open and Cross-platform Trusted Execution Environment”. In: *USENIX Annual Technical Conference*. 2022 (cit. on p. 54).

[386] W. Li et al. “TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms”. In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2019. doi: 10.1145/3313808.3313810 (cit. on p. 54).

[387] L. G. Esswood. “CheriOS: Designing an Untrusted Single-Address-Space Capability Operating System Utilising Capability Hardware and a Minimal Hypervisor”. PhD thesis. Apollo - University of Cambridge Repository, 2020 (cit. on pp. 61, 68, 75, 95, 98, 159, 174).

[388] H. Xia et al. “CheriRTOS: A Capability Model for Embedded Devices”. In: *IEEE 36th International Conference on Computer Design (ICCD)*. 2018. doi: 10.1109/ICCD.2018.00023 (cit. on pp. 64, 95, 97).

[389] Cambridge University. *QEMU-CHERI*. url: https://github.com/CTSRD-CHERI/qemu (cit. on p. 64).

[390] Cambridge University. *TestRIG: Testing Processors with Random Instruction Generation*. url: https://github.com/CTSRD-CHERI/TestRIG (cit. on pp. 64, 102, 145).

[391] H. Xia et al. “CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety”. In: *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*. 2019. doi: 10.1145/3352460.3358288 (cit. on pp. 65, 99, 100, 144).

[392] N. Wesley Filardo et al. “Cornucopia: Temporal Safety for CHERI Heaps”. In: *IEEE Symposium on S&P*. 2020. doi: 10.1109/SP40000.2020.00098 (cit. on pp. 68, 99, 100).

[393] N. W. 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*. 2024. doi: 10.1145/3620665.3640416 (cit. on pp. 68, 99, 100, 144).

[394] R. N. Watson, P. G. Neumann, J. Woodruff, et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Tech. rep. UCAM-CL-TR-987. University of Cambridge, Computer Laboratory, 2023. doi: 10.48456/tr-987. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf (cit. on pp. 68, 74, 86, 89, 136, 140, 141, 159, 162, 174).

[395] F. A. Fuchs, J. Woodruff, et al. “Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V”. In: (2021) (cit. on pp. 68, 108).

[396] P. Metzger et al. “Deprivileging Low-Level GPU Drivers Efficiently with User-Space Processes and CHERI Compartments”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2025. doi: 10.1145/3719027.3765036 (cit. on p. 69).

[397] CTSRD-CHERI. *SPECint2006*. url: https://ctsrd-cheri.github.io/morello-early-performance-results/performance-analysis-spec/specint-2006-benchmark-suite.html (cit. on p. 69).

[398] B. Sá and S. Pinto. *Bao-CHERI: A RISC-V Open-Source Hypervisor with CHERI Support*. Submitted to IEEE Access. 2026 (cit. on p. 71).

[399] B. Sá et al. *RISC-V Virtualization: A Case Study on the CVA6*. Poster presented at RISC-V Summit Europe 2023, Barcelona, Spain. 2023 (cit. on p. 71).

[400] L. Valente et al. “A Heterogeneous RISC-V Based SoC for Secure Nano-UAV Navigation”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2024). doi: 10.1109/TCSI.2024.3359044 (cit. on p. 71).

[401] B. Sá and S. Pinto. *Bao-CHERI: A Pure-Capability RISC-V Hypervisor*. Poster presented at RISC-V Summit Europe 2026, Bologna, Italy. 2026 (cit. on p. 71).

[402] S. Harris et al. “Rust for Morello: Always-On Memory Safety, Even in Unsafe Code”. In: *37th European Conference on Object-Oriented Programming (ECOOP)*. 2023. doi: 10.4230/LIPIcs.ECOOP.2023.39 (cit. on pp. 73, 94, 95).

[403] Google. *Memory safety*. url: https://www.chromium.org/Home/chromium-security/memory-safety/ (cit. on pp. 73, 136, 159).

[404] Y. Iqbal et al. “Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool”. In: *International Conference on Cyber Warfare and Security (ICCWS)*. 2021. doi: 10.1109/ICCWS53234.2021.9703043 (cit. on p. 74).

[405] C.-C. Lin et al. “Support of MISRA C++ Analyzer for Reliability of Embedded Systems”. In: *ACM Transactions Cyber-Physical Systems* (2025) (cit. on pp. 74, 136).

[406] G.-R. Wang and P.-S. Chen. “A GCC-based Compliance Checker for Single-translation-unit, Identifier-related MISRA-C Rules”. In: *Workshop Proceedings of the 49th International Conference on Parallel Processing*. 2020. doi: 10.1145/3409390.3409396 (cit. on p. 74).

[407] A. Al-Boghdady, K. Wassif, et al. “The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT’s Low-End Devices”. In: *Sensors* 21.7 (2021). issn: 1424-8220. doi: 10.3390/s21072329. url: https://www.mdpi.com/1424-8220/21/7/2329 (cit. on p. 74).

[408] S. Peisert. “Unsafe at Any Clock Speed: The Insecurity of Computer System Design, Implementation, and Operation”. In: *IEEE S&P* (2022). doi: 10.1109/MSEC.2021.3127086 (cit. on pp. 74, 136).

[409] X. Zheng et al. “A Closer Look at the Security Risks in the Rust Ecosystem”. In: *ACM Transactions Software Engineering and Methodology* (2023). doi: 10.1145/3624738 (cit. on p. 74).

[410] R. N. Watson, J. Clarke, P. Sewell, et al. *Early performance results from the prototype Morello microarchitecture*. Tech. rep. UCAM-CL-TR-986. Computer Laboratory, University of Cambridge, 2023. doi: 10.48456/TR-986. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.html (cit. on pp. 75, 90–92, 103).

[411] R. N. Watson, P. G. Neumann, J. Woodruff, et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture*. Tech. rep. UCAM-CL-TR-850. University of Cambridge, Computer Laboratory, 2014. doi: 10.48456/tr-850. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-850.pdf (cit. on p. 75).

[412] U. Parliament. *The Industrial Strategy Challenge Fund*. 2021. url: https://committees.parliament.uk/work/1006/the-industrial-strategy-challenge-fund/ (cit. on p. 75).

[413] *Members Archive - CHERI Alliance*. 2024. url: https://cheri-alliance.org/member/ (cit. on p. 75).

[414] D. S. Fowler et al. *Economic factors in CHERI adoption (ResAuto Deliverable D3.2)*. 2024. url: https://wrap.warwick.ac.uk/id/eprint/189379/ (cit. on p. 75).

[415] E. A. Feustel. "The Rice Research Computer: A Tagged Architecture". In: *Proceedings of the Spring Joint Computer Conference*. 1971. doi: 10.1145/1478873.1478920 (cit. on p. 76).

[416] D. Hagimont et al. "Hidden software capabilities". In: *Proceedings of 16th International Conference on Distributed Computing Systems*. 1996. doi: 10.1109/ICDCS.1996.507926 (cit. on p. 77).

[417] J. Rosenberg and D. Abramson. "MONADS-PC-a Capability-Based Workstation to Support Software Engineering". In: *Proceedings of the Hawaii International Conference on System Science*. 1985 (cit. on pp. 78, 79).

[418] D. A. Abramson and J. Rosenberg. "The Microarchitecture of a Capability-Based Computer". In: *Proceedings of the 19th Annual Workshop on Microprogramming*. 1986. doi: 10.1145/19530.19546 (cit. on pp. 78, 79).

[419] A. DeHon et al. "Preliminary Design of the SAFE Platform". In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. 2011. doi: 10.1145/2039239.2039245 (cit. on p. 78).

[420] N. P. Carter, S. W. Keckler, and W. J. Dally. "Hardware Support for Fast Capability-Based Addressing". In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1994. doi: 10.1145/195473.195579 (cit. on pp. 79, 80).

[421] D. D. Redell. *Naming and Protection in Extendable Operating Systems*. Tech. rep. MIT/LCS/TR-140. University of California, 1974 (cit. on p. 79).

[422] P. M. Hansen et al. "A Performance Evaluation of the Intel iAPX 432". In: *SIGARCH Computer Architecture News* (1982). doi: 10.1145/641542.641545 (cit. on p. 80).

[423] S. Mazor. "Intel's 8086". In: *IEEE Annals of the History of Computing* (2010). doi: 10.1109/MAHC.2010.22 (cit. on p. 80).

[424] G. Parmer and R. West. "Predictable Interrupt Management and Scheduling in the Composite Component-Based System". In: *Real-Time Systems Symposium*. 2008, pp. 232–243. doi: 10.1109/RTSS.2008.13 (cit. on pp. 81, 82).

[425] Google Inc. *About Fuchsia*. 2022. url: https://fuchsia.dev/fuchsia-src/concepts (cit. on pp. 81, 82).

[426] seL4 Community. *CapDL | seL4 docs*. 2020. url: https://docs.sel4.systems/projects/capdl/ (cit. on p. 83).

[427] R. N. M. Watson et al. *CHERI C/C++ Programming Guide*. en. UCAM-CL-TR-947. 2020. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.html (cit. on pp. 85, 94).

[428] R. N. Watson et al. "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture". In: *IEEE Micro* (2016). doi: 10.1109/MM.2016.84 (cit. on p. 85).

[429] R. N. Watson, P. G. Neumann, J. Woodruff, et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Tech. rep. UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, 2019. doi: 10.48456/tr-927. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf (cit. on p. 87).

[430] C. Kwan, Y. Xin, et al. "CHERI Concentrate in ACL2: Extended Abstract at the 18th International Workshop on the ACL2 Theorem Prover and Its Applications". In: *Electronic Proceedings in Theoretical Computer Science* (2023). doi: 10.4204/EPTCS.393.2 (cit. on pp. 87, 88).

[431] A. Joannou. "High-performance Memory Safety - Optimizing the CHERI Capability Machine". PhD thesis. University of Cambridge, 2017. doi: 10.17863/CAM.21472 (cit. on pp. 87, 88).

[432] Cambridge University. *BERI*. url: https://github.com/CTSRD-CHERI/beri (cit. on pp. 87, 89, 90).

[433] M. Kaufmann, J. S. Moore, et al. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. isbn: 0792377443 (cit. on p. 88).

[434] A. Joannou et al. "Efficient Tagged Memory". In: *IEEE International Conference on Computer Design (ICCD)*. 2017. doi: 10.1109/ICCD.2017.112 (cit. on pp. 88, 139, 146, 153).

[435] Cambridge University. *CHERI Ibex*. url: https://github.com/CTSRD-CHERI/ibex (cit. on pp. 89, 90, 154, 155).

[436] Cambridge University. *CHERI Piccolo*. url: https://github.com/CTSRD-CHERI/Piccolo (cit. on pp. 89, 90, 154, 155).

[437] Microsoft. *CHERIoT*. url: https://github.com/microsoft/cheriot-ibex (cit. on pp. 89, 90, 154, 155).

[438] S. Amar et al. "CHERIoT: Complete Memory Safety for Embedded Devices". In: *Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture*. 2023. doi: 10.1145/3613424.3614266 (cit. on pp. 89–92, 99, 100).

[439] Marton Bognar. *Proteus: A Configurable RISC-V core*. url: https://github.com/cap-tee/proteus (cit. on pp. 89, 90, 154, 155).

[440] R. N. M. Watson et al. *Bluespec Extensible RISC Implementation: BERI Hardware Reference*. Tech. rep. UCAM-CL-TR-868. University of Cambridge, 2015. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-868.pdf (cit. on pp. 89, 90).

[441] J. Woodruff. *CHERI: A RISC capability machine for practical memory safety*. Tech. rep. UCAM-CL-TR-858. University of Cambridge, Computer Laboratory, 2014. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf (cit. on p. 90).

[442] Arm Ltd. *Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture*. 2022 (cit. on pp. 89, 90, 92).

[443] Zero-Day Labs. *CHERI-CVA6*. url: https://github.com/zero-day-labs/cheri-cva6 (cit. on pp. 89–91, 107, 159).

[444] Microsoft. *Portmeirion*. url: https://www.microsoft.com/en-us/research/project/portmeirion/ (cit. on p. 89).

[445] S. A., T. Chen, D. Chisnall, et al. *CHERIoT Architecture specification*. 0.6 (draft). Microsoft. 2025. url: https://cheriot.org/cheriot-sail/cheriot-architecture.pdf (cit. on p. 89).

[446] K. Nienhuis et al. "Rigorous Engineering for Hardware Security: Formal Modelling and Proof in the CHERI Design and Implementation Process". In: *IEEE Symposium on S&P*. 2020. doi: 10.1109/SP40000.2020.00055 (cit. on pp. 89, 99, 102).

[447] R. Grisenthwaite et al. "The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System". In: *IEEE Micro* (2023). doi: 10.1109/MM.2023.3264676 (cit. on p. 91).

[448] lowRISC. *Sonata System*. url: https://github.com/lowRISC/sonata-system (cit. on p. 91).

[449] lowRISC. *Sonata PCB*. url: https://github.com/newaetech/sonata-pcb/ (cit. on p. 91).

[450] lowRISC CIC, DSbD/UKRI, and N. Technology. *Sunburst Project*. url: https://www.sunburst-project.org/ (cit. on p. 91).

[451] lowRISC. *Sunburst Chip*. url: https://github.com/lowRISC/sunburst-chip (cit. on p. 91).

[452] University of Cambridge. *Compiler Explorer*. 2017. url: https://cheri-compiler-explorer.cl.cam.ac.uk/noscript/z/1Panf4 (cit. on p. 93).

[453] S. Tsampas, D. Devriese, and F. Piessens. "Temporal Safety for Stack Allocated Memory on Capability Machines". In: *IEEE 32nd Computer Security Foundations Symposium (CSF)*. 2019. doi: 10.1109/CSF.2019.00024 (cit. on pp. 93, 99, 100).

[454] AbsInt. *SECOMP*. url: https://github.com/secure-compilation/CompCert (cit. on p. 93).

[455] A. El-Korashy et al. "CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle". In: *IEEE 34th Computer Security Foundations Symposium (CSF)* (2020) (cit. on p. 93).

[456] S. Amar et al. *CapablePtrs*. url: https://gitlab.mpi-sws.org/FCS/capabilities-compilation-public (cit. on p. 93).

[457] Arm Ltd. *Morello Development Tools*. 2022. url: https://developer.arm.com/documentation/109399 (cit. on p. 93).

[458] AbsInt. *CompCert*. url: https://github.com/AbsInt/CompCert (cit. on p. 93).

[459] I. Dudina and I. Stark. "Static analysis to Make the Most of CHERI C/C++ for Existing Code: Improving Memory Safety at Scale". In: *International Journal on Software Tools for Technology Transfer* (2025). doi: 10.1007/s10009-025-00781-6 (cit. on p. 94).

[460] R. N. M. Watson et al. "CHERI: Hardware-Enabled C/C++ Memory Protection at Scale". In: *IEEE Security & Privacy* (2024). doi: 10.1109/MSEC.2024.3396701 (cit. on p. 94).

[461] A. Richardson. "Complete Spatial Safety for C and C++ Using CHERI Capabilities". PhD thesis. University of Cambridge, 2020 (cit. on p. 94).

[462] B. Davis. *CheriABI: Hardware Enforced Memory Safety For FreeBSD*. 2019. url: https://papers.freebsd.org/2019/bsdcan/davis-cheriabi/ (cit. on p. 94).

[463] U. of Cambridge. *Adversarial CHERI Exercises and Missions*. url: https://github.com/CTSRD-CHERI/cheri-exercises (cit. on p. 94).

[464] I. Dudina and I. Stark. "Static Analysis for Transitioning to CHERI C/C++". In: *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 2024. doi: 10.1145/3652588.3663323 (cit. on p. 94).

[465] K. Memarian et al. "Into the Depths of C: Elaborating the de Facto Standards". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016. doi: 10.1145/2908080.2908081 (cit. on p. 94).

[466] K. Memarian et al. "Exploring C Semantics and Pointer Provenance". In: *Proceedings of the ACM on Programming Languages* (2019). doi: 10.1145/3290380. url: https://doi.org/10.1145/3290380 (cit. on p. 94).

[467] K. Memarian. *The Cerberus C Semantics*. University of Cambridge, 2022. url: https://books.google.pt/books?id=ohwS0AEACAAJ (cit. on p. 94).

[468] Cambridge University. *Cerberus*. url: https://github.com/rems-project/cerberus/tree/master (cit. on p. 94).

[469] V. Zaliva et al. “Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2024. doi: 10.1145/3617232.3624859 (cit. on p. 94).

[470] V. Zaliva et al. *CHERI C semantics as an extension of the ISO C17 standard*. Tech. rep. UCAM-CL-TR-988. University of Cambridge, Computer Laboratory, 2023. doi: 10.48456/TR-988 (cit. on p. 94).

[471] L. Cordeiro, B. Fischer, and J. Marques-Silva. “SMT-Based Bounded Model Checking for Embedded ANSI-C Software”. In: *IEEE/ACM International Conference on Automated Software Engineering*. 2009. doi: 10.1109/ASE.2009.63 (cit. on p. 94).

[472] ESBMC. *The ESBMC model checker*. url: https://github.com/esbmc/esbmc (cit. on p. 94).

[473] F. Braße et al. “ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022. doi: 10.1145/3533767.3543289 (cit. on p. 94).

[474] S. H. Park, R. Pai, and T. Melham. “A Formal CHERI-C Semantics for Verification”. In: *Proceedings of 29th International Conference for Tools and Algorithms for the Construction and Analysis of Systems*. 2023. doi: 10.1007/978-3-031-30823-9_28 (cit. on p. 94).

[475] S. H. Park. “A Formal CHERI-C Memory Model”. In: *Archive of Formal Proofs* (2022). issn: 2150-914x (cit. on p. 94).

[476] N. W. S. Sim. *Strengthening Memory Safety in Rust: Exploring CHERI Capabilities for a Safe Language*. 2020. url: https://nw0.github.io/cheri-rust.pdf (cit. on p. 94).

[477] K. Wang et al. *On CHERIfying Linux 5.15*. 2022 (cit. on pp. 95, 96).

[478] H. Almatary, A. Mazzinghi, et al. *Case Study: Securing MMU-less Linux Using CHERI*. 2024. arXiv: 2310.00933 [cs.OS] (cit. on pp. 95, 96).

[479] S. Amar, T. Chen, D. Chisnall, et al. “CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices”. In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 2025. doi: 10.1145/3731569.3764844 (cit. on pp. 95, 97).

[480] P. Olivier, A. Barbalace, and B. Ravindran. “The Case for Intra-Unikernel Isolation”. In: *The 10th Workshop on Systems for Post-Moore Architectures*. 2020 (cit. on pp. 95, 97).

[481] G. Heiser and M. Atkins. “seL4 on Arm Morello”. seL4 Summit 2023. 2023. url: https://sel4summit2023.sched.com/event/1O3je/sel4-on-arm-morello (cit. on pp. 95, 98).

[482] NSA Center for Assured Software. *Juliet C/C++ 1.3 - NIST Software Assurance Reference Dataset*. 2017. url: https://samate.nist.gov/SARD/test-suites/112 (cit. on p. 96).

[483] H. Xia. “Capability Memory Protection for Embedded Systems”. PhD thesis. University of Cambridge, Computer Laboratory, 2021. doi: 10.48456/tr-955. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-955.pdf (cit. on p. 97).

[484] lowRISC CIC, DSbD/UKRI, et al. *CAP-TEE Project*. url: https://cap-tee.org/ (cit. on pp. 97, 108).

[485] *CVE-2020-10064*. 2021. url: https://www.cve.org/CVERecord?id=CVE-2020-10064 (cit. on p. 97).

[486] *NVD - CVE-2020-10064*. 2021. url: https://nvd.nist.gov/vuln/detail/CVE-2020-10064 (cit. on p. 97).

[487] *CVE-2020-10065*. 2021. url: https://www.cve.org/CVERecord?id=CVE-2020-10065 (cit. on p. 97).

[488] *NVD - CVE-2020-10065*. 2021. url: https://nvd.nist.gov/vuln/detail/CVE-2020-10065 (cit. on p. 97).

[489] D. Chisnall. *CHERIoT Programmers' Guide*. 2025. url: https://cheriot.org/book/ (cit. on p. 97).

[490] J. A. Kressel, H. Lefeuvre, and P. Olivier. “Software Compartmentalization Trade-Offs with Hardware Capabilities”. In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. 2023. doi: 10.1145/3623759.3624550 (cit. on p. 97).

[491] J. A. Kressel. *Exploring Software Compartmentalisation with Hardware Capabilities*. 2023 (cit. on p. 97).

[492] H. Lefeuvre et al. “FlexOS: Towards Flexible OS Isolation”. In: *Proceedings of the 27th ACM International Conference on ASPLOS*. 2022. doi: 10.1145/3503222.3507759 (cit. on p. 97).

[493] S. Kuenzer, V. Badoiu, H. Lefeuvre, et al. “Unikraft: fast, specialized unikernels the easy way”. In: *Proceedings of the 16th European Conference on Computer Systems*. 2021. doi: 10.1145/3447786.3456248 (cit. on p. 97).

[494] P. Olivier. “FlexCap: Exploring Hardware Capabilities in Unikernels and Flexible Isolation Oses”. CHERITech'24 presentation. 2024. url: https://flexcap-project.github.io/assets/cheritech24-presentation.pdf (cit. on p. 97).

[495] H. Almatary. *GitHub - CTSRD-CHERI/seL4: The seL4 microkernel*. 2024. url: https://github.com/CTSRD-CHERI/seL4/ (cit. on p. 98).

[496] J. Bramley et al. “Picking a CHERI Allocator: Security and Performance Considerations”. In: *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*. 2023. doi: 10.1145/3591195.3595278 (cit. on p. 98).

[497] J. Evans. “A Scalable Concurrent malloc(3) Implementation for FreeBSD”. In: *FreeBSD* (2006) (cit. on p. 98).

[498] P. Liétar et al. “Snmalloc: A Message Passing Allocator”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. 2019. doi: 10.1145/3315573.3329980 (cit. on p. 98).

[499] D. Lea. *A Memory Allocator*. 1996. url: https://gee.cs.oswego.edu/dl/html/malloc.html (cit. on p. 98).

[500] D. Jacob and J. Singer. “Capability Boehm: Challenges and Opportunities for Garbage Collection with Capability Hardware”. In: *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2022. doi: 10.1145/3516807.3516823 (cit. on pp. 99, 100).

[501] L. Skorstengaard, D. Devriese, and L. Birkedal. “Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management”. In: *ACM Transactions on Programming Languages and Systems* (2019). doi: 10.1145/3363519 (cit. on pp. 99, 100).

[502] A. L. Georges et al. “Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities”. In: *Proceedings of the ACM on Programming Languages* (2021). doi: 10.1145/3434287 (cit. on pp. 99, 100).

[503] A. L. Georges, A. Trieu, and L. Birkedal. “Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities”. In: *Proceedings of the ACM on Programming Languages* (2022). doi: 10.1145/3527318 (cit. on pp. 99, 100).

[504] Vercammen, T. and Strydonck, T. V. and others. “Borrowed Capabilities: Flexibly Enforcing Revocation on a Capability Architecture”. In: *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2021, pp. 80–85. doi: 10.1109/EuroSPW54576.2021.00015 (cit. on pp. 99, 100).

[505] G. Merve, E. Håkan, et al. *Mon CHÈRI <3 Adapting Capability Hardware Enhanced RISC with Conditional Capabilities*. 2024. arXiv: 2407.08663 [cs.CR]. url: https://arxiv.org/abs/2407.08663 (cit. on pp. 99, 100).

[506] L. Skorstengaard, D. Devriese, and L. Birkedal. “StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities”. In: *Proceedings of the ACM on Programming Languages* (2019). doi: 10.1145/3290332 (cit. on p. 100).

[507] D. Lowther et al. “Secure Scripting with CHERIoT MicroPython”. In: *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. 2025. doi: 10.1145/3708493.3712694 (cit. on p. 101).

[508] George Robotics Limited. *MicroPython - Python for microcontrollers*. 2024. url: https://micropython.org/ (cit. on p. 101).

[509] Q. Ochem, R. Tice, et al. *Ada for Embedded C Developer*. Tech. rep. AdaCore, 2021 (cit. on p. 101).

[510] *libreSSL*. 2024. url: https://www.libressl.org/ (cit. on p. 101).

[511] *libssh2*. 2024. url: https://libssh2.org/ (cit. on p. 101).

[512] H. Almatary. "CHERI Compartmentalisation for Embedded Systems". PhD thesis. Apollo - University of Cambridge Repository, 2022. doi: 10.17863/CAM.88674 (cit. on p. 101).

[513] V. A. Sartakov et al. "CAP-VMs: Capability-Based Isolation and Sharing in the Cloud". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2022 (cit. on p. 101).

[514] V. A. Sartakov et al. "ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities". In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2023 (cit. on p. 102).

[515] J. You, Y. Chu, and L. Zhao. "Accelerating Cold Start of Thread-level Sandbox Using Snapshot and tfork". In: *IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*. 2024. doi: 10.1109/ICPADS63350.2024.00046 (cit. on p. 102).

[516] CheriBSD community. *CheriBSD Packages*. 2024. url: https://pkg.cheribsd.org/ (cit. on pp. 102, 103).

[517] Arm. *Morello Platform Model Reference Guide Revision: Version 2.0*. 2021. url: https://developer.arm.com/documentation/102225/ (cit. on p. 102).

[518] Cambridge University. *Sail Morello*. url: https://github.com/CTSRD-CHERI/sail-morello (cit. on p. 102).

[519] Cambridge University. *Sail CHERI-RISC-V*. url: https://github.com/CTSRD-CHERI/sail-cheri-riscv (cit. on p. 102).

[520] Cambridge University. *Sail CHERI-MIPS*. url: https://github.com/CTSRD-CHERI/sail-cheri-mips (cit. on p. 102).

[521] A. L. Duque Antón et al. "VeriCHERI: Exhaustive Formal Security Verification of CHERI at the RTL". In: *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 2025 (cit. on p. 102).

[522] A. Mazzinghi, R. Sohan, and R. N. M. Watson. "Pointer Provenance in a Capability Architecture". In: *Proceedings of the 10th USENIX Conference on Theory and Practice of Provenance*. 2018 (cit. on p. 102).

[523] C. Community. *GitHub - CTSRD-CHERI/gdb: The GNU debugger extended to support CHERI*. 2025. url: https://github.com/CTSRD-CHERI/gdb (cit. on p. 102).

[524] V. Safronov et al. “How Memory-Safe is IoT? Assessing the Impact of Memory-Protection Solutions for Securing Wireless Gateways”. In: *Proceedings of the 14th International Conference on the Internet of Things*. 2025. doi: 10.1145/3703790.3703820 (cit. on p. 103).

[525] M. Dodson et al. “CHERI Macaroons: Efficient, Host-Based Access Control for Cyber-Physical Systems”. In: *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020. doi: 10.1109/EuroSPW51379.2020.00099 (cit. on p. 104).

[526] A. Birgisson et al. “Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud”. In: *Network and Distributed System Security Symposium*. 2014 (cit. on p. 104).

[527] M. Dodson. “Capability-Based Access Control for Cyber Physical Systems”. PhD thesis. Apollo - University of Cambridge Repository, 2021. doi: 10.17863/CAM.77125. url: https://www.repository.cam.ac.uk/handle/1810/329679 (cit. on p. 104).

[528] K. Cross. “Investigating the Feasibility of Using CHERI-Enabled Arm Morello Boards for Cloud-Based Trusted Execution”. PhD thesis. Jesus College, 2023 (cit. on p. 104).

[529] D. Ferraro, A. Bastoni, et al. “Enabling Security on the Edge: A CHERI Compartmentalized Network Stack”. Design, Automation and Test in Europe Conference (DATE) 25 - To appear. 2025 (cit. on p. 104).

[530] J. Bean, M. Ryland, M. S. Wilson, et al. *The Security Design of the AWS Nitro System*. Tech. rep. Amazon Web Services, 2024. url: https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf (cit. on p. 104).

[531] CAMB. *CAMB (Cloud Attestables on Morello Boards)*. 2021. url: https://www.cl.cam.ac.uk/research/srg/projects/camb/ (cit. on p. 104).

[532] B. Connectivity et al. *AutoCHERI*. url: https://autocheri.tech/ (cit. on p. 104).

[533] N. Hoang Nga, S. Siraj Ahmed, K. Emirhan, et al. “Addressing Automotive Cybersecurity Risks with an ARM Morello Capability-Enhanced Prototype”. Preprint, Authorea. 2024. url: 10.22541/au.171401582.21766451/v1 (cit. on p. 105).

[534] C. S. Kalutharage, S. Mohan, et al. “Enhancing Automotive Intrusion Detection Systems with Capability Hardware Enhanced RISC Instructions-Based Memory Protection”. In: *Electronics* (2025). doi: 10.3390/electronics14030474 (cit. on p. 105).

[535] D. Ferraro, B. Sá, J. Martins, et al. *Security, Safety, and Predictability of CHERI RISC-V for Drone Systems*. Poster presented at RISC-V Summit Europe 2024, Munich, Germany. 2024. url: https://riscv-europe.org/summit/2024/media/proceedings/posters/96_poster.pdf (cit. on p. 105).

[536] PX4. *Open Source Autopilot for Drones - PX4 Autopilot*. 2024. url: https://px4.io/ (cit. on p. 105).

[537] NVIDIA. *CUDA, release: 10.2.89*. 2020. url: https://developer.nvidia.com/cuda-toolkit (cit. on p. 105).

[538] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science and Engineering* (2010) (cit. on p. 105).

[539] Y. Guo, Z. Zhang, and J. Yang. "GPU Memory Exploitation for Fun and Profit". In: *33rd USENIX Security Symposium*. 2024 (cit. on p. 105).

[540] J. Roels, A. Jacobs, and S. Volckaert. "CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World". In: *Proceedings of the 18th European Workshop on Systems Security*. 2025. doi: 10.1145/3722041.3723099 (cit. on p. 105).

[541] S.-O. Park et al. "Mind Control Attack: Undermining Deep Learning with GPU Memory Exploitation". In: *Computers and Security* (2021). doi: 10.1016/j.cose.2020.102115 (cit. on p. 105).

[542] F. D. Pustelnik, X. M. Sass, and J.-P. Seifert. "Whispering Pixels: Exploiting Uninitialized Register Accesses in Modern GPUs". In: *IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. 2024. doi: 10.1109/EuroSP60621.2024.00026 (cit. on p. 105).

[543] M. Naylor, A. Joannou, A. T. Markettos, et al. *CHERI-SIMT report: implementing capability memory protection in GPGPUs*. Tech. rep. UCAM-CL-TR-997. University of Cambridge, 2025. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-997.html (cit. on p. 106).

[544] Jenkinson, G. and Mazzinghi, A. and others. *CHERI-based Memory Protection and Compartmentalisation for Web Services on Morello*. Tech. rep. Capabilities Limited, 2024. url: https://github.com/Capabilities-Limited/Capabilities-Limited.github.io/blob/main/pdfs/20231020-capltd-cheri-server-evaluation-report-public-version1-FINAL.pdf (cit. on p. 106).

[545] Watson, R. N.M. and Laurie, B. and others. *Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem*. Tech. rep. Capabilities Limited, 2021. url: https://github.com/Capabilities-Limited/Capabilities-Limited.github.io/blob/main/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf (cit. on p. 106).

[546] P. Zalewski. *Baking Android for ARM Morello without Morello – The Good Penguin*. 2022. url: https://www.thegoodpenguin.co.uk/blog/baking-android-for-arm-morello-without-morello/ (cit. on p. 106).

[547] A. T. Markettos et al. "Position Paper:Defending Direct Memory Access with CHERI Capabilities". In: *Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2021. doi: 10.1145/3458903.3458910 (cit. on p. 107).

[548] FreeBSD Community. *bhyve - FreeBSD Wiki*. 2024. url: https://wiki.freebsd.org/bhyve (cit. on p. 107).

[549] H. D. Chirammal, P. Mukhedkar, et al. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016 (cit. on p. 107).

[550] T. Van Strydonck et al. "CHERI-TrEE: Flexible Enclaves on Capability Machines". In: *IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 2023. doi: 10.1109/EuroSP57164.2023.00070 (cit. on pp. 107, 108).

[551] C. Community. *bhyve hypervisor - Getting Started with CheriBSD 24.05*. 2024. url: https://ctsrd-cheri.github.io/cheribsd-getting-started/features/bhyve.html (cit. on p. 107).

[552] T. Cloosters et al. "Memory Corruption at the Border of Trusted Execution". In: *IEEE Security & Privacy* (2024). doi: 10.1109/MSEC.2024.3381439 (cit. on p. 108).

[553] J. Jennifer, J. Minmin, et al. "CHERI-Crypt: Transparent Memory Encryption on Capability Architectures". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2025) (cit. on p. 108).

[554] J. Z. Yu et al. "Capstone: A Capability-based Foundation for Trustless Secure Memory Access". In: *32nd USENIX Security Symposium*. 2023 (cit. on p. 108).

[555] M. Lipp et al. "Meltdown: reading kernel memory from user space". In: *Communications of the ACM* (2020). doi: 10.1145/3357033 (cit. on p. 108).

[556] Q. Ge et al. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* (2018). doi: 10.1007/s13389-016-0141-6 (cit. on pp. 108, 133).

[557] F. A. Fuchs, J. Woodruff, et al. "Safe Speculation for Cheri". In: *IEEE 42nd International Conference on Computer Design (ICCD)*. 2024. doi: 10.1109/ICCD63220.2024.00063 (cit. on p. 108).

[558] F. A. Fuchs, J. Woodruff, et al. "Architectural Contracts for Safe Speculation ". In: *2023 IEEE 41st International Conference on Computer Design (ICCD)*. 2023. doi: 10.1109/ICCD58817.2023.00093 (cit. on p. 108).

[559] R. Watson, J. Woodruff, et al. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Tech. rep. UCAM-CL-TR-916. University of Cambridge, 2018. url: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf (cit. on p. 108).

[560] J. Henes, M. Muench, et al. "Transient-execution attacks on the CHERI Morello platform". In: *Proceedings of the Microarchitecture Security Conference* (2025). doi: 10.46586/uasc.2025.104 (cit. on p. 109).

[561] M. Bechtel and H. Yun. “Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019. doi: 10.1109/RTAS.2019.00037 (cit. on pp. 111, 158).

[562] N. Flaherty. *Europe Steps up as RISC-V Ships 10bn Cores*. 2022. url: https://www.eenewseurope.com/en/europe-steps-up-as-risc-v-ships-10bn-cores/ (cit. on p. 111).

[563] M. Gautschi et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2017). doi: 10.1109/TVLSI.2017.2654506 (cit. on p. 111).

[564] L. Valente et al. “HULK-V: a Heterogeneous Ultra-low-power Linux capable RISC-V SoC”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023. doi: 10.23919/DATE56975.2023.10137252 (cit. on p. 111).

[565] D. Rossi et al. “Vega: A Ten-Core SoC for IoT Endnodes With DNN Acceleration and Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode”. In: *IEEE Journal of Solid-State Circuits* (2022). doi: 10.1109/JSSC.2021.3114881 (cit. on p. 111).

[566] A. Garofalo et al. “Darkside: 2.6GFLOPS, 8.7mW Heterogeneous RISC-V Cluster for Extreme-Edge On-Chip DNN Inference and Training”. In: *IEEE 48th European Solid State Circuits Conference (ESSCIRC)*. 2022. doi: 10.1109/ESSCIRC55480.2022.9911384 (cit. on p. 111).

[567] F. Zaruba and L. Benini. “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology”. In: *IEEE Transactions on Very Large Scale Integration Systems* (2019). doi: 10.1109/TVLSI.2019.2926114 (cit. on pp. 111, 112, 116, 117, 137, 140).

[568] C. Chen et al. “Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product”. In: *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020. doi: 10.1109/ISCA45697.2020.00016 (cit. on p. 111).

[569] F. Ficarelli et al. “Meet Monte Cimone: exploring RISC-V high performance compute clusters”. In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. 2022. doi: 10.1145/3528416.3530869 (cit. on p. 111).

[570] A. Waterman, K. Asanović, and J. Hauser. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. Tech. rep. Document Version 1.12-draft. RISC-V Internacional, 2022 (cit. on pp. 111, 114).

[571] A. Patel et al. “Embedded Hypervisor Xvisor: A Comparative Analysis”. In: *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015. doi: 10.1109/PDP.2015.108 (cit. on pp. 112, 132).

[572] S. Zhao. “Trap-less Virtual Interrupt for KVM on RISC-V”. In: *KVM Forum*. 2020 (cit. on p. 112).

[573] G. Heiser. “seL4 is verified on RISC-V!” In: *RISC-V International*. 2020 (cit. on p. 112).

[574] J.-Y. Hwang et al. “Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones”. In: *5th IEEE Consumer Communications and Networking Conference*. 2008. doi: 10.1109/ccnc08.2007.64 (cit. on p. 112).

[575] R. Ramsauer et al. “Static Hardware Partitioning on RISC-V: Shortcomings, Limitations, and Prospects”. In: *IEEE 8th World Forum on Internet of Things (WF-IoT)*. 2022. doi: 10.1109/WF-IoT54382.2022.10152063 (cit. on p. 112).

[576] J. Andersson. “Development of a NOEL-V RISC-V SoC Targeting Space Applications”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020. doi: 10.1109/DSN-W50199.2020.00020 (cit. on pp. 112, 133).

[577] N. Gala et al. “SHAKTI Processors: An Open-Source Hardware Initiative”. In: *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems (VLSID)*. 2016. doi: 10.1109/VLSID.2016.130 (cit. on p. 112).

[578] J. Scheid and J. Scheel. *RISC-V "stimecmp / vstimecmp"Extension Document Version 0.5.4-3f9ed34*. 2021. url: https://github.com/riscv/riscv-time-compare/releases/download/v0.5.4/Sstc.pdf (cit. on pp. 112, 116).

[579] A. Baum et al. *RISC-V Base Cache Management Operation ISA Extensions Document Version 1.0-fd39d01*. 2022. url: https://raw.githubusercontent.com/riscv/riscv-CMOs/master/specifications/cmobase-v1.0.pdf (cit. on p. 117).

[580] J. Hauser. *The RISC-V Advanced Interrupt Architecture Document Version 0.3.0-draft*. 2022. url: https://github.com/riscv/riscv-aia (cit. on pp. 118, 132).

[581] S. Venkata et al. “SD-VBS: The San Diego Vision Benchmark Suite”. In: *IEEE International Symposium on Workload Characterization*. 2009. doi: 10.1109/IISWC.2009.5306794 (cit. on pp. 123, 137, 148).

[582] T. Kloda et al. “Lazy Load Scheduling for Mixed-criticality Applications in Heterogeneous MPSoCs”. In: *ACM Transactions Embedded Computing Systems* (2023). doi: 10.1145/3587694 (cit. on p. 123).

[583] J. Balkind et al. “OpenPiton: An Open Source Manycore Research Framework”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 2016. doi: 10.1145/2872362.2872414 (cit. on p. 132).

[584] K. Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, 2016 (cit. on p. 133).

[585] Gaisler. *GRLIB IP Core Users Manual*. 2022. url: https://www.gaisler.com/products/grlib/grip.pdf (cit. on p. 133).

[586] Y.-J. Chang and M.-F. Lan. “Two New Techniques Integrated for Energy-Efficient TLB Design”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2007). doi: 10.1109/TVLSI.2006.887813 (cit. on p. 134).

[587] B. Pham et al. “CoLT: Coalesced Large-Reach TLBs”. In: *45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012. doi: 10.1109/MICRO.2012.32 (cit. on p. 134).

[588] M. Talluri et al. “Tradeoffs in Supporting Two Page Sizes”. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. 1992. doi: 10.1145/139669.140406 (cit. on p. 134).

[589] B. Pham et al. “Increasing TLB reach by exploiting clustering in page translations”. In: *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014. doi: 10.1109/HPCA.2014.6835964 (cit. on p. 134).

[590] G. Cox and A. Bhattacharjee. “Efficient Address Translation for Architectures with Multiple Page Sizes”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 2017. doi: 10.1145/3037697.3037704 (cit. on p. 134).

[591] J. H. Ryoo et al. “Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017. doi: 10.1145/3079856.3080210 (cit. on p. 134).

[592] G. Kandiraju and A. Sivasubramaniam. “Going the Distance for TLB Prefetching: An Application-Driven Study”. In: *Proceedings 29th Annual International Symposium on Computer Architecture*. 2002. doi: 10.1109/ISCA.2002.1003578 (cit. on p. 134).

[593] A. Bhattacharjee, D. Lustig, and M. Martonosi. “Shared Last-Level TLBs for Chip Multiprocessors”. In: *IEEE 17th International Symposium on High-Performance Computing Architecture*. 2011. doi: 10.1109/HPCA.2011.5749717 (cit. on p. 134).

[594] S. Bharadwaj et al. “Scalable Distributed Last-Level TLBs Using Low-Latency Interconnects”. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture*. 2018. doi: 10.1109/MICRO.2018.00030 (cit. on p. 134).

[595] T. W. Barr, A. L. Cox, and S. Rixner. “Translation Caching: Skip, Don’t Walk (the Page Table)”. In: *SIGARCH Computer Architecture News* (2010). doi: 10.1145/1816038.1815970 (cit. on p. 134).

[596] R. Bhargava et al. “Accelerating Two-Dimensional Page Walks for Virtualized Systems”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2008. doi: 10.1145/1346281.1346286 (cit. on p. 134).

[597] C. che Tsai, D. E. Porter, and M. Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *USENIX Annual Technical Conference*. 2017 (cit. on p. 136).

[598] A. Vahldiek-Oberwagner et al. “ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK)”. In: *Proceedings of the 28th USENIX Security Symposium*. 2019 (cit. on p. 136).

[599] S. Gal-On. *CoreMark Benchmark*. EEMBC (Embedded Microprocessor Benchmark Consortium). 2009. url: https://www.eembc.org/coremark/ (cit. on pp. 137, 148).

[600] M. Guthaus et al. “MiBench: A Free, Commercially Representative Embedded Benchmark Suite”. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*. 2001. doi: 10.1109/WWC.2001.990739 (cit. on pp. 137, 148).

[601] Cambridge University. *CHERI CAP LIB*. url: https://github.com/CTSRD-CHERI/cheri-cap-lib (cit. on p. 141).

[602] Cambridge University. *RVFI-DII*. url: https://github.com/CTSRD-CHERI/TestRIG/blob/master/RVFI-DII.md (cit. on p. 145).

[603] D. Du and B. Yang. *RISC-V S-level Physical Memory Protection (SPMP)*. 2025. url: https://github.com/riscv/riscv-spmp/blob/main/rv-spmp-spec.pdf (cit. on pp. 159, 175).

[604] A. Khan, D. Xu, and D. J. Tian. “EC: Embedded Systems Compartmentalization via Intra-Kernel Isolation”. In: *IEEE Symposium on Security and Privacy (SP)*. 2023. doi: 10.1109/SP46215.2023.10179285 (cit. on p. 159).