*Technical Report*

UCAM-CL-TR-1001
ISSN 1476-2986

Number 1001

**UNIVERSITY OF CAMBRIDGE**

**Computer Laboratory**

# Toward transient-execution attack mitigations on CHERI

Franz A. Fuchs

August 2025

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

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

This technical report is based on a dissertation submitted November 2024 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville & Caius College.

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

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

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-1001*

# Toward transient-execution attack mitigations on CHERI

Franz Anton Fuchs

## Abstract

This thesis explores how to protect Capability Hardware Enhanced RISC Instructions (CHERI) systems from transient-execution attacks. Transient-execution attacks shocked the computing world as they allow security mechanisms to be circumvented via seemingly safe performance-enhancing mechanisms. These attacks use misguided speculation to access secrets and transmit them via a side channel. Since the initial discovery of this attack class, every year saw fresh attacks being discovered with a lack of mitigation mechanisms.

CHERI defines architectural capabilities that help to tackle spatial and temporal memory safety issues. However, the CHERI ISA has not been designed with transient-execution vulnerabilities in mind. In order to satisfy performance requirements, CHERI implementations employ out-of-order and speculative execution mechanisms. The lack of ISA-level guarantees leads to multiple attack scenarios on conventional and CHERI systems. This thesis demonstrates a full-scale attack on CHERI-Toooba that manages to break CHERI's security guarantees in speculation and reliably leaks a secret value. Motivated by these findings, I developed ISA-level contracts that restrict speculative execution for both conventional and CHERI systems. As a major contribution, these contracts close a gap in architectures by giving guarantees about speculation mechanisms, which allows secure software to be built atop these contracts. I evaluated my contracts on CHERI-Toooba, which is an out-of-order, superscalar implementation of CHERI-RISC-V. My contracts offer substantial security guarantees and can surprisingly lead to improvements in both cycle performance as well as area usage on FPGAs.

Making fine-grained compartmentalisation robust against transient-execution attacks is critical for the overall security of CHERI systems. In this work, I compare multiple different solutions and introduce Thread ID Capability (TIDC) registers as a fast and compact solution to compartmentalisation on CHERI-RISC-V. In order to facilitate fast and secure transitions between compartments, this work finds that microarchitectures must not allow microarchitectural state to leak. This thesis suggests multiple approaches to separating microarchitectural state and evaluates their performance on multiple compartmentalisation models. Last, this thesis introduces the notion of Compartment ID (CID) sealing. In this work, I implement a necessary subset of this approach that allows out-of-order microarchitectures to keep track of current software-defined CIDs. I find this approach to have significant performance advantages over conventional speculation fences.

# Acknowledgements

I would like to thank my supervisor Prof. Simon W. Moore for his guidance and support throughout the journey that led to this dissertation. Simon, you have believed in me and gave me a chance where not many others would have done so. I am beyond grateful for what you have done for me.

Being part of the CHERI team is an exceptional experience. I would like to thank Prof. Robert N. M. Watson who drives the overall CHERI project. I am truly indebted to every single past and current member of the team. Especially, I would like to thank Dr Jonathan Woodruff, Dr Peter Rugg, Dr Alexandre Joannou, Jessica Clarke, Konrad Witaszczyk, Alfredo Mazzinghi, Dapeng Gao, Samuel Stark, Edward Napierala, Dr Hesham Almatary, and Tom Bytheway. Thank you for always having been available to discuss my proposals, pointing out their shortcomings, and motivating me to go beyond what I thought I could reach.

Furthermore, I would like to thank my examiners Prof Timothy Jones and Prof David Oswald for their invaluable feedback and suggested improvements.

The journey to this thesis would not have been possible without the tireless support of the many dear people in my life. First and most importantly: Mama, Papa, Eli und Anna – Ihr seid schon immer für mich da gewesen und habt mich auf allen Wegen unterstützt. Nur durch euch ist das alles möglich. Konrad and Alfredo: Not only have you taught me a lot about CHERI software, but also you have become very dear friends who never fail to understand and support me. Cambridge is a wonderful place and I will truly miss it: Especially through the German Society and Caius it has become a second home as I thought I would never have it. So many dear people are the reason to this and there is not enough space to thank all of them. I will try nevertheless: Tobia, Greta, Nico, Dante, Greg, Felix, Beyza, Marek, Dominik, Elke, Navid, Adrian, Lukas, Natasha, Tobias, David, Marcela, Francesca, and Ryan. Each of you has played their part in me succeeding in my PhD and having one of the greatest times of my life.

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-18-C-0016 ("ECATS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This work was supported by the Engineering Physical Sciences Research Council (EPSRC) [EP/S030867/1]. This work was supported in part by the DoD Information Analysis Center Program Management Office (DoD IAC PMO), sponsored by the Defense Technical Information Center (DTIC) under Contract No. FA807518D0004. Any opinions,

findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Installation Contracting Agency (AFICA).

# Contents

# List of Figures

# Glossary

**Application Binary Interface (ABI)**

An interface between binaries. 51, 139

**Architectural Compartment Identifier (ACID)**

Architectural representation of a Compartment Identifier (CID). 150, 152, 154

**Arithmetic Logic Unit (ALU)**

Unit responsible for conducting simple arithmetic calculations, e.g., a logic OR. 39, 70, 111, 112, 122, 123, 127, 148, 154, 164

**Address Space Identifier (ASID)**

Number identifying an address space, which is used to implement virtual memory. 149

**Address Space Layout Randomisation (ASLR)**

Security mechanism of randomising at which start addresses code, data, heap, and other segments are laid out that is commonly implement operating systems to mitigate attacks. 41

**Access System Register (ASR)**

A permission bit in CHERI capabilities that defines access for CSR and SCR registers. 108, 109, 110

**Berkeley Packet Filter (BPF)**

A network packet filter, which allows user space processes to specify which packets they want to receive. 53

**Block Random Access Memory (BRAM)**

Modules on FPGAs that allow to access data via a limited number of read and write ports. 136, 137, 140

**Branching Speculation Contract (BSC)**

An architectural contract constraining control-flow speculation. 58, 60, 61, 62, 64, 67, 68, 69, 70, 71, 72, 73, 74, 130

**Branch Target Buffer (BTB)**

Microarchitectural unit used for predicting targets of control-flow operations. 41, 43, 59, 60, 61, 62, 64, 67, 68, 70, 71, 72, 82, 87, 88, 94, 131, 134, 136, 140, 143, 145, 161

**Compartment Identifier (CID)**

Number identifying a compartment at a certain time in a certain space in the system. 17, 30, 132, 150, 151, 152, 153, 154, 155, 156, 157, 163, 164, 165

**Capability Speculation Contract (CSC)**

An architectural contract constraining speculation around CHERI capabilities. 15, 78, 79, 80, 81, 82, 83, 84, 86, 87, 88, 89, 92, 93, 94, 95, 97, 98

**Control and Status Register (CSR)**

A set of registers that changes how the system currently operates, e.g., address translation. 68, 108, 111, 112, 114, 115, 120, 127, 137, 142, 143, 147, 154, 156

**Default Data Capability (DDC)**

CHERI capability used for data accesses in legacy instructions. 45, 105, 106, 121

**Direct Memory Access (DMA)**

The direct access of peripherals to main memory in order to facilitate them to function without the CPU. 55

**First In First Out (FIFO)**

A data structure ensuring elements are dequeued in the order they have been enqueued. 71

**Field Programmable Gate Array (FPGA)**

A device containing configurable logic that allows re-programming in the field. 21, 49, 67, 97, 135, 136, 137, 157

**Floating Point Unit (FPU)**

Unit responsible for conducting floating point arithmetic calculations. 123, 127, 148

**Hardware Description Language (HDL)**

Language to define the behaviour of a circuit. 48

**Hardware Performance Monitoring (HPM)**

A set of counters implemented in hardware that keep track of events, e.g., speculatively executed instructions. 65

**Invoked Data Capability (IDC)**

Data capability used when invoking a pair of capabilities to transition to a different compartment. 46, 168, 169

**Instruction-Set Architecture (ISA)**

Definition of instructions, registers, and other necessary guarantees for a machine. 26, 27, 37, 38, 39, 41, 43, 44, 45, 46, 47, 50, 51, 53, 54, 56, 59, 62, 63, 65, 79, 83, 101, 102, 103, 104, 105, 108, 112, 121, 122, 127, 145, 167, 168, 169

**Instruction-Set Simulator (ISS)**

A piece of software that simulates the architecture in software instead of implementing it in hardware. 29

**Lookup Table (LUT)**

Logic element on an Field Programmable Gate Array (FPGA) that can be programmed to represent any logic function. 93, 97, 126, 140, 156

**Microarchitectural Data Sampling (MDS)**

Type of transient-execution attacks that relies on wrongly, speculatively forwarding values that can be made visible via a side channel. 40, 42, 64

**Memory Ordering Speculation Contract (MOSC)**

An architectural contract constraining the speculative ordering of memory instructions. 61, 63, 64, 69, 70

**Program Counter (PC)**

Pointer to the currently executed instruction. 21, 39, 45, 59, 67, 69, 79, 87, 111

**Program Counter Capability (PCC)**

CHERI capability extension of the Program Counter (PC). 45, 46, 79, 87, 88, 94, 95, 96, 97, 98, 104, 105, 150, 151, 153, 154, 155, 156, 167, 168, 169

**Pattern History Table (PHT)**

Microarchitectural structure that stores the most recently taken and non-taken decisions of direct branches. 38, 59

**Process Identifier (PID)**

Number commonly implemented by operating systems to identify processes running currently on the system. 149, 150

**Point of No Return (PONR)**

Point in the processor pipeline where an instruction is guaranteed to commit. 78, 80

**Partition and Tag Identifier (PTID)**

Number representing the current identifier used for partitioning or tagging microarchitectural state. 17, 132, 133, 134, 135, 136, 137, 138, 142, 154, 155

**Re-Order Buffer (ROB)**

Microarchitectural structure that is used to keep track of instructions in flight on an out-of-order processor. 86, 93, 111, 112, 123, 127

**Return Stack Buffer (RSB)**

Microarchitectural unit used for predicting targets of operations returning from subroutines. 41, 60, 62, 68, 71, 72, 74, 75, 82, 87, 134, 135, 161

**RISC-V Formal Interface (RVFI)**

A standard architectural tracing format used for RISC-V implementations. 65

**Software Compartment Identifier (SCID)**

Software representation of a Compartment Identifier (CID), which might be a larger space than the architecturally available bits. 150, 152

**Special Capability Register (SCR)**

Capability-wide registers that are used as control and status registers for CHERI systems. 108, 110, 112, 120, 122

**Thread Identifier (TID)**

Architectural mechanism for software to reliably identity the current software thread it is running on. 108, 109, 110, 113, 114, 115, 127, 137

**Translation Lookaside Buffer (TLB)**

Security mechanism of randomising at which start addresses code, data, heap, and other segments are laid out that is commonly implement operating systems to mitigate attacks. 42, 49

**Thread-Local Storage (TLS)**

Memory that is dedicated to per-thread data. 109

**Translation Speculation Contract (TSC)**

An architectural contract constraining speculative translation of virtual to physical addresses. 57, 63, 64, 66, 67, 69, 70

# Chapter 1

# Introduction

This thesis covers transient-execution attack mitigations on CHERI-RISC-V systems. Transient-execution attacks are a class of attacks that exfiltrate secrets from processors by carefully exploiting speculative execution. CHERI (Capability Hardware Enhanced RISC Instructions) is a security extension that defines unforgeable pointers called capabilities. Previous work has shown that CHERI processors can be susceptible to transient-execution attacks [34, 35]. This has motivated the research on mitigating transient-execution attacks presented in this dissertation.

## 1.1 The necessity and curse of microarchitectural speculation

In early stages of many text books on computer architecture, one will find the concept of *pipelining* explained [71]. These pipelining models remain on simple – often five-stage long – pipelines, but soon the topic of pipeline hazards is covered. Among them are *control hazards*, which arise when a direct branch enters the pipeline, but it is not yet known whether the branch is *taken* (jumping to the offset in the instruction) or *not taken* (continuing at the next instruction). As a solution to avoid introducing pipeline bubbles, these text books often introduce *branch prediction*, which is the attempt to predict in which direction a branch goes. Because there is no certainty to whether the prediction is true, the instructions following the branch are executed *speculatively*. Instructions that should have never executed, but did due to misspeculation, are *squashed* by the microarchitecture to ensure they have no architectural effect.

With the need for more performance in every processor generation, the processor designs became more sophisticated and the amount of speculation increased in order to keep the pipelines of the processor filled. Many of today's microarchitectures attempt to squeeze out the last remaining bit of instruction-level parallelism by speculating. Instructions are

executed *out-of-order* often hundreds of instructions before they would appear in program order and it may be very deep in the pipeline that an instruction has to be squashed.

Speculative execution can have drawbacks, as demonstrated by the Intel Pentium 4 processor. Its pipeline and prediction mechanisms were not coordinated resulting in major performance penalties [61]. Just less than two decades after, the security world saw a new type of negative consequences for excessive speculation. The *Spectre* [58] and *Meltdown* [64] attacks managed to violate security policies during speculative execution and thus managed to access secret values. In these attacks, instructions that never appeared in the control-flow of the original sequential program are executed and able to leak secret information.

These two attacks have sparked research into new follow-up attacks that have been published on a regular basis [74]. The discovery of Spectre and Meltdown has given rise to a difficult problem for both industry and academia. Current microarchitectures inherently rely on speculation to reach performance goals, but this speculation poses a security risk. This becomes even more critical when looking at secure architectures. Often microarchitectures implementing secure Instruction-Set Architecture (ISA) extensions are based on conventional microarchitectures. This is also the case for CHERI on Arm processors. The Morello CPU, which is an Arm implementation of the abstract CHERI protection model, is based on an industrial-scale processor that is widely used in non-security contexts [8].

## 1.2 Hypotheses

In this section, I present the hypotheses I evaluate in this thesis. At the end of each chapter, I will provide a conclusion to the chapter and which hypotheses were evaluated.

**Hypothesis H.1**

Conventional processor architectures can be extended with architectural speculation contracts to constrain overall speculation.

Previous work on speculation contracts has shown the potential of architectural speculation contracts, but practical application is lacking. The challenges of defining constraints on speculation at the architectural level are not yet known. This includes being able to test if microarchitectures are compliant against the contracts.

**Hypothesis H.2**

A conventional microarchitecture can be extended to implement the speculation contracts and does not incur significant performance overheads.

Defining architectural contracts is an important step that needs to be complemented by a proof-of-concept implementation. The existence of speculation contracts can only be justified if they can be implemented with a sensible overhead. A part of the architecture that cannot be implemented in modern microarchitectures with good performance will remain a useless piece of architecture. It is important, therefore, to conduct an implementation on a superscalar, out-of-order machine.

**Hypothesis H.3**

The CHERI architecture can be extended using architectural speculation contracts that constrain the behaviour of microarchitecture in speculation in order to avoid information leakage.

CHERI is a large and abstract security ISA extension, which can be mapped to different base ISAs. It is crucial to evaluate whether CHERI can be composed with ISA-level contracts as well as whether useful architectural properties can be established. This includes that CHERI microarchitectures can be tested against the new contracts.

**Hypothesis H.4**

A CHERI microarchitecture can be extended to implement the architectural speculation contracts and does not incur significant Performance, Power, and Area (PPA) overheads.

Capability contracts need to be implemented in a superscalar, out-of-order microarchitecture in order to prove that their existence is justified. If this is not possible, I must revisit the definition of the aforementioned contracts.

**Hypothesis H.5**

CHERI-RISC-V can support library compartmentalisation, previously only available on the Morello platform.

In conventional systems, transient-execution attacks within a protection domain are a threat because protection domains are usually large and contain multiple parts that do not fully trust each other, e.g., a malicious library in a process can leak information from the main program. With fine-grained compartmentalisation, only code that fully trusts each other is in the same compartment. Thus, compartmentalisation is an option to express boundaries that can be defended against transient-execution attacks as they exist on conventional systems. Currently, library-compartmentalisation is the only widely supported compartmentalisation model on CHERI. This model runs only on the Morello platform and makes use of architectural features like register banking used widely on Arm, but not on RISC-V. Therefore, it is important to research which primitives the CHERI-RISC-V platform needs to support this compartmentalisation model.

**Hypothesis H.6**

CHERI allows the definition of hardware compartments defined in the architecture and can be implemented to have a small impact on PPA metrics.

In order to support the developer's intended location of boundaries, the architecture needs to provide primitives to communicate domain boundaries from software to hardware. These primitives need to be defined in the architecture as well as implemented in hardware to prove the feasibility of the approach.

**Hypothesis H.7**

CHERI hardware compartments can offer robustness against transient-execution attacks by implementing confidentiality and integrity, thus mitigating transient-execution attacks.

CHERI hardware-level compartmentalisation offers high security guarantees against memory safety vulnerabilities. In order to mitigate transient-execution attacks, it is necessary to guarantee both confidentiality and integrity of hardware-level compartments.

## 1.3 Attacker model

These hypotheses span a wide field of topics. In this section, I present the limits of the scope in which my research took place.

I consider power analysis, e.g., as described by Kocher et al. [59] and any side-channel attack that needs physical access to the hardware – as for example optical, acoustic, or electromagnetic side-channel attacks – out-of-scope. In order to prevent attackers physically accessing hardware, other measures that are orthogonal to my research need to be in place. In general, every attack that is a consequence of the attacker undertaking physical attacks on the hardware will not be considered within this dissertation. This includes attacks at the foundry level, e.g., injecting trojans [65]. Moreover, I expect the underlying physical hardware to work as specified. Physical hardware failures where software is able to permanently change parts of hardware state, which it has no access to, e.g., as demonstrated by the *rowhammer* [56] attacks are not considered within the scope this doctoral research. These failures are on the physical level of abstraction and are not within my threat model.

Transient-execution attacks can be split into two parts: accessing a secret value and leaking this value through a side channel. In this work, I will focus on the mechanisms of accessing a secret value. This approach is powerful and practical because without access to a secret value an attacker cannot leak it. Furthermore, trying to close all known side channels leaves open the opportunity for attackers to construct a new side channel.

Last, every attack that is due to misconfiguration is out-of-scope. For example, a bug in the operating system might allow userspace processes to access more memory than intended, therefore allowing information leakage. To be more explicit, if the attack runs successfully on an Instruction-Set Simulator (ISS), it is also allowed to run successfully on any microarchitecture because the failure is not caused by the microarchitecture.

## 1.4 Contributions

This section lists the contributions of this work.

In Chapter 3, I design contracts for safe speculation in conventional architectures. This chapter is an extension of *Architectural Contracts for Safe Speculation*, which was accepted at ICCD 2023. In this chapter, I define exemplar architectural speculation contracts and develop a portable testing strategy, which flags multiple violations. I co-design contract enforcement mechanisms and show that they can be implemented with near-zero overhead in SPECint2006.

In Chapter 4, I extend the idea of contracts to CHERI capabilities. This chapter is a superset of material presented in *Safe Speculation for CHERI*, which was accepted at ICCD 2024. After forming the Capability Speculation Contract (CSC), I develop a test generator, which reproduces all known Meltdown-CF violations in CHERI-Toooba. After assembling a full-scale Meltdown-CF exploit showing the severity of the attack, I co-design a mechanism for complete CSC enforcement in CHERI-Toooba, leading to moderate performance overheads.

In Chapter 5, I contribute to safe and lightweight compartmentalisation in CHERI. The outcome of this research leads to an extension in the effort to standardise CHERI within RISC-V. In this chapter, I survey existing CHERI compartmentalisation primitives and find them not suitable for current CHERI compartmentalisation within RISC-V. I develop and implement a set of primitives, and run a compartmentalisation approach within the CheriBSD operating system using these primitives.

In Chapter 6, I protect the microarchitectural state of compartments against each other. I design an approach that compartmentalises state in prediction structures in CHERI-Toooba using both tagging and partitioning of resources. This approach is refined in Chapter 7, in which I add security measures for domain crossings on CHERI systems. After showing that barrier instructions remain insufficient for performance goals, I design and implement Compartment Identifier (CID) based sealing, which demonstrates the negligible performance impact of this approach.

## 1.5 Open-source contributions

During my PhD studies, I have made the following contributions to the open-source community:

**CHERI-Toooba** (Chapters 3, 4, 5, 6, and 7) All the microarchitectural analysis and evaluation carried out in the course of my doctoral research was done on CHERI-Toooba, a CHERI enhanced superscalar, out-of-order processor developed by many others. Among others, my contributions include the implementation of TID/TIDC, fast general-purpose and floating-point register clearing, partitioning and tagging support for prediction units, and Compartment ID sealing. Furthermore, I have added Konata[1] tracing to CHERI-Toooba. `https://github.com/CTSRD-CHERI/Toooba`

**TestRIG and QuickCheckVEngine** (Chapters 3 and 4) The implementation of testing CHERI-Toooba against the speculation contracts formulated in this thesis was carried out on TestRIG and the default test generation engine QuickCheckVEngine. Furthermore, I did bug fixes and minor improvements at various places in both repositories. `https://github.com/CTSRD-CHERI/TestRIG` and `https://github.com/CTSRD-CHERI/QuickCheckVEngine`

**Sail CHERI-RISC-V** (Chapter 5) I implemented the work on TID/TIDC in the Sail formal ISA model of CHERI-RISC-V. Furthermore, I carried out some minor bug fixes. `https://github.com/CTSRD-CHERI/sail-cheri-riscv`

**QEMU** (Chapter 5) I implemented the work on TID/TIDC in QEMU, which enabled the fast bring-up of library compartmentalisation on CHERI-RISC-V. `https://github.com/CTSRD-CHERI/qemu`

**RISC-V HPM Events** (Chapters 3 and 4) I created this repository to have an improved implementation of Hardware Performance Monitoring (HPM) events commonly used in the Bluespec-based cores from the University of Cambridge. `https://github.com/CTSRD-CHERI/RISCV_HPM_Events`

**RISC-V CHERI** (Chapter 5) I am a co-author of the document resulting from this repository. The current state of this is a draft specification, which is currently under decision to be ratified as an extension of the RISC-V architecture. I contributed the *Ztid* extension to this document and provided bug fixes in various places. `https://github.com/riscv/riscv-cheri`

**CHERI Specification** (Chapters 4, 5, and 7) I am a co-author of this open-source specification. I added multiple sections in the *Experimental CHERI extensions* chapter and provided bug fixes in various places in the document. `https://github.com/CTSRD-CHERI/cheri-specification`

[1] See GitHub repository: `https://github.com/shioyadan/Konata`.

## 1.6 Publications

I have authored or co-authored the following publications during my PhD studies:

**Safe Speculation for CHERI** (Chapter 4) (*42nd IEEE International Conference on Computer Design, November 2024*) I am the first author of this paper [36]. Full author list: Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Alexandre Joannou, Jessica Clarke, John Baldwin, Brooks Davis, Peter G. Neumann, Robert N. M. Watson, Simon W. Moore.

**Randomized Testing of RISC-V CPUs using Direct Instruction Injection** (Chapters 3 and 4) (*IEEE Design & Test, Volume 41, Issue 1, February 2024*) I have co-authored this paper and contributed to the engineering of the test framework as well as writing and editing the paper's text [50]. Full author list: Alexandre Joannou, Peter Rugg, Jonathan Woodruff, Franz A. Fuchs, Marno van der Maas, Matthew Naylor, Michael Roe, Robert N. M. Watson, Peter G. Neumann, Simon W. Moore.

**Architectural Contracts for Safe Speculation** (Chapter 3) (*41st IEEE International Conference on Computer Design, November 2023*) I am the first author of this paper [37]. I pioneered the ideas and methods behind the paper as well as carrying out the major parts of the implementation and evaluation. Full author list: Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Marno van der Maas, Alexandre Joannou, Alexander Richardson, Jessica Clarke, Nathaniel Wesley Filardo, Brooks Davis, John Baldwin, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson.

**Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)** (Chapters 4, 5, and 7) (*Technical Report, University of Cambridge, September 2023*) I authored a section in the part about compartment IDs in this technical report [93]. Furthermore, I provided minor editing when I ran across inconsistencies in the technical report. Full author list: Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, Hongyan Xia.

**RISC-V Specification for CHERI Extensions** (Chapter 5) (*RISC-V Specifications, not yet published*) I co-authored the underlying report of this specification and helped in meetings preparing this specification. I filed multiple issues and pull requests on the GitHub repository. I am leading the development of the *Ztid* extension for

lightweight compartmentalisation. Current author list[2]: Thomas Aird, Hesham Almatary, Andres Amaya Garcia, John Baldwin, Paul Buxton, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Franz A. Fuchs, Timothy Hutt, Alexandre Joannou, Martin Kaiser, Tariq Kurd, Ben Laurie, Marno van der Maas, Maja Malenko, A. Theodore Markettos, David McKay, Jamie Melling, Stuart Menefy, Simon W. Moore, Peter G. Neumann, Robert Norton, Alexander Richardson, Michael Roe, Peter Rugg, Peter Sewell, Carl Shaw, Ricki Tura, Robert N. M. Watson, Toby Wenman, Jonathan Woodruff, Jason Zhijingcheng Yu

Furthermore, I am a co-author on the following papers that are planned to be submitted after the submission date of this thesis:

**Superscalar CHERI: implementing capabilities in sophisticated microarchitectures** The topic of this paper is how to implement CHERI in superscalar, out-of-order microarchitectures on the example of CHERI-Toooba. I contributed development and improvement of parts in CHERI-Toooba, discussing ideas and approaches, and editorial feedback.

**Fast and fine-grained compartmentalisation on CHERI** The topic is this paper will be work on hardware compartmentalisation laid out in Chapters 5, 6, 7, and software compartmentalisation done by others. Potentially, this will be split into multiple papers.

## 1.7 Presentations

I have given or am planning to give the following talks and presentations during my PhD studies:

**Safe Speculation for CHERI** (*42nd IEEE International Conference on Computer Design, November 2024, Milan, Italy*) I presented my paper on proposing and evaluating architectural speculation contracts for CHERI systems, which includes a full-scale transient-execution attack. Website: `https://www.iccd-conf.com/Home.html`

**Architectural Contracts for Safe Speculation** (*41st IEEE International Conference on Computer Design, November 2023, Washington D.C., USA*) I presented my paper on proposing and evaluating architectural constraints for speculative execution in order to be able to mitigate transient-execution attacks. Website: `https://iccd-conf.com/2023/Home.html`

[2]This is version v0.9.4.

**Can CHERI Stop Transient-Execution Attacks?** (*RISC-V Summit Europe, June 2023, Barcelona, Spain*) This presentation showed the clear need for architectural solutions in the RISC-V ISA in order to be able to mitigate transient-execution attacks and proposed solutions for it. Website: `https://riscv-europe.org/summit/2023/`

**Toward transient-execution attack mitigations on CHERI compartments** (*CHERI Technical Workshop, March 2023, Glasgow, United Kingdom*) I explained how transient-execution attacks work on CHERI systems and proposed microarchitectural compartmentalisation as a solution. Website: `https://www.dcs.gla.ac.uk/~jsinger/cheritech23.html`

**Demonstrating Memory-safety Features under CheriBSD on a Multi-core, Superscalar Softcore** (*RISC-V Spring Week, May 2022, Paris, France*) I presented a demonstration how programming for CHERI works and how the technology can mitigate attacks targeting memory safety of programs. Website: `https://open-src-soc.org/2022-05/index.html`

## 1.8 Posters

I have prepared or helped to prepare the following posters:

**The RISC-V CHERI Security Extension** (*RISC-V Summit North America, October 2024, Santa Clara, CA, USA*) I contributed in parts to the engineering framework and the research presented on the poster. Website: `https://events.linuxfoundation.org/riscv-summit/`

**Fast CHERI-RISC-V Compartmentalisation** (*RISC-V Summit Europe, June 2024, Munich, Germany*) I presented the advancements of my research on CHERI-RISC-V compartmentalisation. Website: `https://riscv-europe.org/summit/2024/`

**Mitigating Transient-Execution Attacks on CHERI** (*DSbD All Hands Meeting, March 2024, London, UK*) I presented my research on how CHERI compartments can help mitigate transient-execution attacks. Website: `https://www.dsbd.tech/all-hands-march-highlights/`

**CHERI Compartments – Toward Transient-Execution Attack Mitigations on CHERI** (*RISC-V Summit Europe, June 2023, Barcelona, Spain*) This poster accompanied my talk at the summit. I presented challenges and opportunities for CHERI compartments to facilitate mitigating transient-execution attacks. Website: `https://riscv-europe.org/summit/2023/`

**Randomized Testing of RISC-V CPUs using Direct Instruction Injection** (*RISC-V Summit Europe, June 2023, Barcelona, Spain*) I contributed in parts to

the engineering framework and the research presented on the poster. Website: `https://riscv-europe.org/summit/2023/`

**Mitigating Transient-Execution Attacks with CHERI Compartments** (*DSbD All Hands Meeting, April 2023, London, UK*) I presented my research on transient-execution mitigations with CHERI compartments. Website: `https://www.dsbd.tech/event/dsbd-all-hands/`

## 1.9 Demonstrations

For the RISC-V Summit Europe 2024 in Munich[3], I showcased my group's research on CHERI-RISC-V. In this demonstration talk called *RISC-V Summit Europe – CHERI Demo*, I showed the capabilities of the CHERI-RISC-V open source stack. During the presentation, I went through a subset of the *CHERI-Exercises*[4] in order to show the differences between a conventional system and CHERI.

## 1.10 Thesis overview

In Chapter 2, I provide background on the field researched in this dissertation. In Chapter 3, I define and evaluate hardware speculation contracts on conventional systems. I extend these with architectural security guarantees for CHERI systems presented in Chapter 4. In Chapter 5, I research the mechanisms of fast fine-grained compartmentalisation on CHERI and both present and evaluate a sufficient solution for CHERI-RISC-V. In Chapter 6, I show my work on protecting microarchitectural state of compartments against attacks from other compartments. This is complemented by my work on securing domain transitions in CHERI systems, which is presented in Chapter 7. I provide my conclusions to my hypotheses in Chapter 8.

[3] `https://riscv-europe.org/summit/2024/conference`.
[4] `https://github.com/CTSRD-CHERI/cheri-exercises`.

# Chapter 2

# Background

In this chapter, I explain the previous work and concepts in the research field related to my doctoral studies. First, I elaborate on common performance enhancing concepts in modern microarchitectures, followed by transient-execution attacks, defence mechanisms against these attacks, and the newly evolving field of hardware-software contracts. In the remaining sections, I present the CHERI capability model, previous research conducted on CHERI systems when targeted by transient-execution attacks, and implementations of CHERI.

## 2.1 Architectures and microarchitectures

The ISA – often to referred just as *the architecture* – is the contract between software and hardware. It is an abstract model that defines what a processor can do and what the effects of these actions are. Among other things, an ISA defines which registers a machine has as well as which instructions the machine executes and the effects of these instructions. There exist multiple ISAs of which the `x86` family and the Arm architectures remain commercially dominant [75].

On the other side, a microarchitecture is a concrete implementation of an ISA. While an ISA is general, a microarchitecture is often targeted to a specific purpose in order to fulfil the needs of the market, e.g., optimising for performance. Implementing the architecture is the only constraint a microarchitecture is bound to.

## 2.2 Side channels

All transient-execution attacks presented in Section 2.4 need a form of side channel to transmit the speculatively accessed secret. A side channel allows for unintended information flow out of a program. In theory, attackers can use any form of side channel to facilitate transient-execution attacks. Most often, proof-of-concept attacks use a cache-timing side-channel. Generally, transient-execution attacks do not use covert channels. Covert channels are intentional where two protection domains collaborate, whereas a side channel denotes that the victim domain allows its secret to leaked unintentional by an attacking domain [39]. Transient-execution attacks are a sophisticated form of attacks which are not necessary if two protection domains are already collaborating.

There are many timing side channels that have been researched of which cache-timing side-channels remain the most popular. Cache-timing side-channels are based on creating a difference in access time between cache lines. A shorter access time means that a certain cache line has been loaded by code before, and a longer access time means that the cache line is not present, e.g., by having been evicted. Both of these observations allow inferences about secrets handled by code before. There are many cache-timing side-channel techniques of which Prime+Probe [70] and FLUSH+RELOAD [108] are popular examples.

Moreover, other microarchitectural structures can be used to transmit secrets via timing differences, e.g., the Pattern History Table (PHT) [32]. A list of microarchitectural structures used in side-channel attacks is presented by Randal [74].

## 2.3 Speculative and transient execution

Traditionally, microarchitects have been mostly concerned about the big three goals when designing a new processor: Power, performance, and area. For the desktop and server processor market, the ever increasing demand for faster computation resources has led to powerful processors, which use a set of sophisticated performance enhancing techniques explained in the following paragraphs.

All ISAs contain one or more instructions that allow the programmer to branch to another piece code depending on a condition, e.g., a register holding a particular value or a condition code. In order to efficiently use the fetch unit and keep the pipelines filled, implementations want to predict the direction outcome of a branch before actually evaluating it to avoid pipeline bubbles. A common technique for processors is to use knowledge of past instances of this branch, e.g., a branch in a `for` loop, to gain information and use that to predict future instances of this branch. For branch direction prediction, a wide-spread approach is to keep a history of branch instructions. Processors store a number of the past outcomes of an instance of a branch instruction – the *local* history – and an overall history of the last executed branches in the instruction stream – the *global* history. Furthermore, processors

engage in predicting the Program Counter (PC) of an instruction called *branch target prediction*. For this, most processors keep past targets in one or more microarchitectural structures. These structures vary in size and how sophisticated they are, but mostly contain entries of a tag and the jump target itself. Additionally, some processors have dedicated structures for targets of `return` instructions. Other than the traditional speculation on the outcome of branches and jumps, modern microarchitectures increasingly engage in value prediction. One example for value prediction is address analysis to allow reordering of store and load instructions. If a load instruction accesses a different memory location than an older store, the load instruction can be executed first without changing the logic of the program[1]. However memory addresses might not be fully known in earlier pipeline stages and thus modern microarchitectures look into applying address dependence prediction. As Yu et al. [109] propose, many instructions in the ISA can be seen as *implicit branches*. They are not branches declared by the ISA, but in modern implementations they are likely to be treated as branches because speculation is performed on them. An example is the two memory access instructions depicted in this listing:

```
// Pre-condition: the microarchitecture does not yet know
// whether x11 and x13 hold the same address
sd x10, 0(x11)
ld x12, 0(x13)
```

In this example, the microarchitecture will try to predict whether the registers `x11` and `x13` hold the same value. If they do, the store and the load are dependent and the load needs to be executed after the store. However, if they are not dependent, the load instruction can be executed before the store instruction, potentially leading to performance improvements.

Apart from prediction, modern microarchitectures incorporate out-of-order execution as a performance enhancing mechanism. This approach allows a processor to execute younger instructions if one or more older instructions cannot continue due to a dependency, e.g., an arithmetic operation cannot execute due to an outstanding memory load. However, a younger instruction without dependencies can go ahead and execute. This enhances performance and makes more efficient use of microarchitectural resources, e.g., the Arithmetic Logic Unit (ALU) would not be used as frequently otherwise. The ISA does not know about out-of-order execution, but executes one instruction strictly after the other. This means that all instructions take effect – their change on the architectural state – become visible after each other. In out-of-order processors this sequential programming model needs to be maintained. Therefore, these processors have dedicated mechanisms to *commit* an instruction, which is the stage where instructions take architectural effect. Even though instructions can execute out-of-order, they still commit in program order. However, out-of-order execution can lead to microarchitectural side effects. One example would be that a younger instruction has been executed and took microarchitectural effect, but an

[1]This does not hold if there are instructions between the store and load that would forbid reordering, e.g., `fence` instructions.

older instruction throws an exception and therefore the younger instruction should have never been executed in the first place. The younger instruction then has to be *squashed*, thus it must not take architectural effect.

Instructions issued due to out-of-order execution or branch prediction are classified as *speculative* – they possibly do not commit. A speculative instruction becomes *transient*, when it will be squashed, but already took microarchitectural effect, e.g., by pulling in a cache line.

## 2.4 Transient-execution attacks

In early 2018, the publication of Spectre [58] and Meltdown [64] changed the computing world. These two attacks demonstrated that microarchitectural state changes through transiently executed instructions can be used for sophisticated attacks. This led to extensive research into related attacks, respective defence mechanisms, and changed the way the security community argues about architectural state compared to microarchitectural state today.

Transient-execution attacks consist of two major building blocks of which both are essential for an attack to be successful. First, speculative execution will violate a security policy and lead to the access or processing of a secret value. Second, this secret value is transmitted to the attacker via a side-channel. Without the side-channel, the rollback for the misspeculated execution will erase the secret value and make it impossible to retrieve. In this section, I categorise transient-execution attacks by the underlying speculation mechanism that allows the security policy violation.

Transient-execution attacks are split into three sub classes of attacks: Spectre-like attacks, Meltdown-like attacks, and Microarchitectural Data Sampling (MDS) attacks:

### 2.4.1 Spectre attacks

Spectre-like attacks exploit microarchitectural state changes caused by misprediction of control- or data-flow. These attacks target prediction units on a microarchitecture: branch direction prediction, branch target prediction, and value prediction. At the time of writing this thesis, four major variants of Spectre attacks are known [22], where each of these attacks can have multiple flavours.

Spectre-PHT [58] mistrains the history of a specific branch instruction such that it is mispredicted when the attack takes place. This leads to instructions guarded by an `if` statement to be executed speculatively and possibly leak secret information to the attacker through transient microarchitectural state changes.

Next, the Spectre-BTB [58] subclass targets branch target prediction. The goal is to change or completely overwrite an entry in the Branch Target Buffer (BTB) in order to

impact speculative control flow, which can enable an attacker to transiently execute kernel code and thereby leak a secret. In a similar way, Spectre-RSB [60, 66] attacks try to impact control flow by overwriting Return Stack Buffer (RSB) entries. However, overwriting RSB entries leads to a totally different attack vector, e.g., transient return-oriented programming due to the circular RSB implementation most microarchitectures have.

In addition to the two classes seen as distinct attacks, some researchers managed to combine parts from the classes and develop a new one, e.g., the Spectre-BHB attacks which combines branch history and branch target prediction [17].

Spectre-STL (Store-To-Load) [45] targets the misprediction of memory dependencies. Modern processors employ memory disambiguation – predicting independent memory operations – in order to enhance overall performance. When prediction is carefully mislead by the attacker, loads will be issued early enabling an attacker to read stale data from the memory subsystem.

### 2.4.2 Meltdown attacks

Meltdown-like attacks are characterised by microarchitectural state changes caused by transiently executed instructions following a faulting instruction. They are generally caused by permissions checked late in the pipeline. The original Meltdown-US (User-Supervisor) [64] attack accessed a supervisor-only page from user space and made the secret visible through the cache hierarchy. This was caused by processors speculatively allowing the page access before the faulting memory access was microarchitecturally cancelled. Meltdown-US allows attackers to read out kernel memory and use the secrets for further attacks, e.g., Address Space Layout Randomisation (ASLR) information. Not all vendors produced processors vulnerable to Meltdown-US, e.g., AMD states that none of their processors are vulnerable to it [5].

In a similar manner, Meltdown-GP (General Protection) [11] is a subclass of Meltdown-style attacks that targets transiently reading a privileged register from user mode and then making it visible through microarchitectural state, e.g., in order to learn more about the physical mapping of the process.

Meltdown-style attacks come in many variants and are often tailored exactly to a specific ISA, e.g., specifically for Intel processors, Foreshadow [19, 99] attacks target Intel SGX enclaves [46].

### 2.4.3 Microarchitectural data sampling attacks

MDS attacks do not fit in the definitions from above as they are neither based on prediction mechanisms nor on failing instructions. Their goal is to *sample* data from another protection domain or another address space. The base mechanisms are facilitated by microarchitectural components that speculatively forward values to loads such that data appears out of *thin air*. This aggressive forwarding strategy has been implemented as a performance enhancement – especially given the great difference in access times between the core and memory. In comparison to Meltdown-style attacks, MDS attacks do not rely on failing memory accesses, but rather that the memory request is speculatively satisfied by a microarchitectural data structure holding secret data, e.g., speculative value forwarding. There exist different variants of MDS, where each is specialised on another microarchitectural structure. For example, RIDL [84] leaks data from line fill buffers, Fallout [21] from store buffers, and ZombieLoad [85] uses L1 data cache evictions to sample data.

## 2.5 Defences

In this section, I present defence approaches against transient-execution attacks implemented in hardware and in software.

The software solutions were the first security mechanisms to be deployed as the computing world needed to react to the emerge of transient-execution attacks and early solutions could not be postponed to the release of the next generation of processors. For Meltdown-US, the most prominent solution was to unmap kernel pages in user space – called KPTI [27]. However, kernel isolation can cause significant performance overheads, due to Translation Lookaside Buffer (TLB) flushes, different cache use patterns, and multiple other factors [26]. As described in the previous paragraphs, Meltdown-style attacks are due to insufficient or even non-existing checks before allowing instructions to take microarchitectural effect. Therefore, Meltdown-style attacks are considered a problem directly caused by hardware and should be solved by hardware rather than software as demonstrated by AMD [5].

On the other side, Spectre-style attacks can be more considered a problem hardware and software have in common, which led to multiple mitigation mechanisms to be put in place on the software side. To mitigate Spectre-PHT, Intel recommends using the pre-existing `lfence` instruction that stops speculation until all older instructions have committed [47]. Furthermore, WebKit suggests implementing pointer masking [72], which uses an `and` operation in order to prevent the pointer from being increased out of bounds. Similarly, Speculative Load Hardening [23] uses conditional instructions to prevent speculation from continuing. Both pointer masking and Speculative Load Hardening incur significant performance overheads due to cutting off speculation completely where deployed. Furthermore,

it is not trivial for compilers to insert fence instructions or similar mitigation mechanisms as shown by Kocher [57]. For Spectre-BTB and Spectre-RSB, fence instructions can also be used to stop speculation in a similar manner to Spectre-PHT. Specifically for Spectre-BTB, the *retpoline* [48] approach – which avoids making use of the BTB at all – has been developed. An approach that can mitigate all cross protection domain Spectre-like attacks is separation of privileges (see more in Section 2.7), e.g., *Site Isolation* [76]. Spectre-STL has not been in the focus of mitigation mechanisms and therefore has no dedicated solution. Load fences, e.g., as the `lfence` by Intel prevent the load from being issued until all architecturally prior instructions have committed and therefore form a mitigation mechanism.

Hardware approaches can be divided into two sub categories. There exist approaches that want to close a side-channel used by transient-execution attacks and approaches that want to prevent computation on forbidden data in the first place. In the first case, the cache timing side-channel is targeted most by transient-execution attacks and therefore is in the centre of the research, e.g., MuonTrap [1] or InvisiSpec [107]. All approaches have in common that they do not introduce speculatively loaded cache lines into the cache hierarchy and therefore make it impossible for the attacker to retrieve the secret later. However, these approaches only close the cache-timing side channel and the attacker can leak the secret via another side-channel attack. On the other hand, different approaches microarchitecturally forbid the flow of speculatively accessed data used by transient-execution attacks, e.g., Speculative Taint Tracking [109] or NDA [98]. These two and other approaches delay the execution of instructions that depend on secret data. Furthermore, processor vendors have introduced indirect branch speculation barriers into their ISAs [11, 47]. Both approaches incur tremendous performance overhead when used extensively due to stopping speculative execution for many cycles.

## 2.6 Contracts

After initial work on transient-execution attacks and defences, research has been conducted to find out how the hardware-software interface can be improved in order to tackle transient-execution vulnerabilities. The first bits were to define formal semantics for transient-execution attacks and use symbolic execution to prove leakage in real-world programs [42]. This led to the development of formal hardware-software contracts that enable comparison of different implementations' models regarding which contract they comply with [43]. This work was extended by *Revizor* – a technique that allowed a simplified version of the x86 architecture to be tested against a subset of the contracts [68]. However, the contracts research field is in its infancy. The initial foundation has been laid, but the contracts are not yet fully applicable to current architectures as they only look at simplified ISAs and remain incomplete. With Revizor, the hardware-software contracts field received a practical application and performs fundamental research on the

applicability of contract testing. However, their approach relies on heavily guided testing as well as testing for side-channels instead of testing for the actual microarchitectural contract violation.

## 2.7 Compartmentalisation

*Compartment* is an abstract term in computing and has multiple interpretations. In general, we understand compartmentalisation to be a form *privilege decomposition* – an idea that has been existent in computer security for a long time [53, 55, 73, 76, 95]. The main goal is to conduct separation of abstract processes in order to decrease the attack surface. This not only protects against currently known attacks, but also against the ones we are unaware of [93]. Compartmentalisation has multiple forms of instantiation, e.g., having instances of the code for each set of input data known as *horizontal* compartmentalisation compared to dividing the code into different layers known as *vertical* compartmentalisation [44]. This means that different systems likely interpret compartmentalisation in different ways leading to completely different security guarantees for each single compartment. Therefore, it is essential for every compartmentalisation model to precisely define what a compartment means, how it interacts with other compartments, and which security guarantees come with the compartmentalisation model. In this work, I explain the constraints on the implementation of microarchitectural compartments and where the ISA needs to be extended to have a precise model.

Compartmentalisation can be split into software-based and hardware-based compartmentalisation. Earlier work of software-based compartmentalisation includes *Efficient software-based fault isolation* [89], which functions on the basis of separating resources into different segments. An extension of this work is *hardware-assisted fault isolation*, which proposes hardware-level measures to provide *regions* that allow for isolation [67]. One often used and more extreme form of hardware-based compartments are *enclaves*, which want to attain *strong isolation* meaning that all software of one process is entirely separated from all other software. While enclaves have great security properties, they often have tremendous drawbacks that are not acceptable in real-world scenarios, e.g., *MI6* – a RISC-V based enclave implementation – has more than 15% run time overhead and does not scale to multiple active enclaves [18].

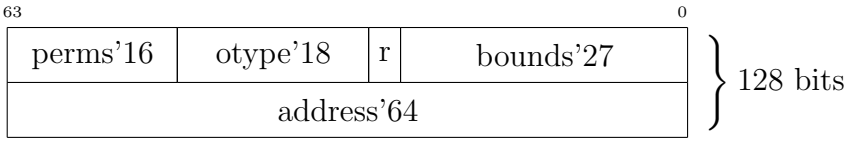



Figure 2.1: Bit representation of a 128-bit capability (excluding the one bit validity tag) with 'r' standing for 'reserved'.

## 2.8 RISC-V

RISC-V is an open-source ISA that sees a lot of interest in both academia and industry, with approximately 1.3 billion RISC-V processors shipped in 2023 [54]. RISC-V is a highly modular ISA with only a small mandatory part that needs to be implemented [29, 30]. There exists a plethora of extensions, which implementors can choose to include, e.g., implementing floating-point support is split into two separate extensions **F** and **D**. RISC-V implementations are denoted by a string that lists all the extensions implemented. For example, `RV64IFDC` is a 64-bit RISC-V processor that implements the **I**nteger, **F**loating, **D**ouble, and **C**ompressed instructions extensions.

## 2.9 CHERI

CHERI (Capability Hardware Enhanced RISC Instructions) [93, 105] is an ISA security extension for unforgeable bounded pointers called *capabilities*. A capability is a fat pointer [49] extending the pointer address with metadata including bounds information and permissions. In CHERI, a capability is twice the size of the address; for example a 64 bit integer pointer becomes a 128 bit capability as shown in Figure 2.1. The integrity of capabilities in CHERI is protected by an additional one bit validity tag, which is stored atomically with the capability it protects. In order to access memory at a certain address, a program needs a valid capability that grants access to the range of memory the address is in. The CHERI model only defines instructions that monotonically decrease privilege, which means that no instruction can increase the privileges authorised by a capability. The CHERI model defines two additional registers. CHERI extends the PC to the Program Counter Capability (PCC) therefore offering protection for instruction memory. Second, the Default Data Capability (DDC) is the capability through that all legacy (non-capability) memory accesses are authorised. On a CHERI system every memory access needs to be authorised by a valid capability. CHERI not only protects against spatial memory-safety attacks, but is also able to mitigate temporal memory-safety attacks to the heap [33, 106].

Furthermore, compartmentalisation is a core feature of the CHERI protection model. In CHERI, a compartment can be implemented as a set of capabilities that are not overlapping. With the help of trusted software or dedicated instructions, compartment switches can

be implemented. A compartment switch is a controlled violation of monotonicity within CHERI. A jump from one compartment to the other will give access to new capabilities, which have previously not been accessible. In CHERI v9, there are two mechanisms available that allow compartment transitions. The primitives for these mechanisms are detailed in Figure 2.2. A *sentry* (sealed entry capability) is a code capability that serves as entry point into a code block. Any manipulation of the sentry – as on any sealed capability – will clear the tag bit and thus make the capability invalid. Therefore, the only useful operation code can do is to jump to a sentry. This jump is an atomic operation that combines unsealing the sentry and installing it in the PCC register. This design ensures that a full compartment jump is done without leaking capabilities from one compartment to another one. A similar concept is a *sealed capability pair*. The pair consists of a sealed code capability and a sealed data capability. The pair is linked by its *otype*. Only if this value matches for both capabilities, then the pair can be atomically unsealed and installed in the respective PCC and Invoked Data Capability (IDC) registers. In Chapter 5, I elaborate on how to use sentries for implementing a sufficient set of primitives supporting compartmentalisation.

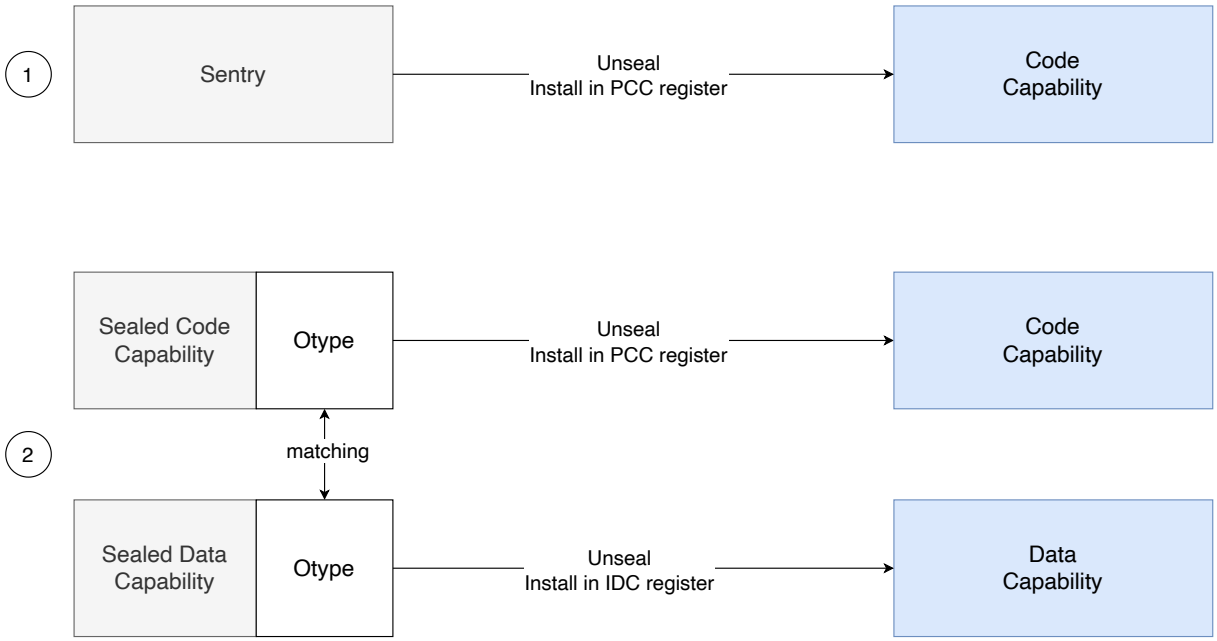



Figure 2.2: CHERI v9 describes two compartmentalisation primitives. The first diagram shows how a compartment transition can be facilitated via a *sentry*. The second diagram illustrates the concept of a *sealed code and data capability pair*. Both approaches work with atomic instructions that ensure that both the unsealing of the capabilities and the actual jump are done in one operation.

The CHERI model described above is abstract in its nature. It can be instantiated as a security extension to multiple conventional ISAs. CHERI has been mapped to MIPS [105], RISC-V [93], and to the Arm v8 architecture [7], with CHERI-RISC-V being the main current open-source research effort. Therefore, CHERI-RISC-V is the focus of my research on CHERI. The CHERI-RISC-V ISA describes all CHERI instructions

mapped to RISC-V, CHERI exceptions, CHERI extended registers, and other features of CHERI on RISC-V. At that time of writing this report, four RISC-V implementations have been extended for CHERI: CHERI-Ibex [97] and CHERI-Piccolo [81] are two 32-bit implementations. CHERI-Flute [80] implements RV64 with a 5-stage in-order pipeline. CHERI-Toooba [104] is a superscalar, out-of-order processor implementing 64-bit RISC-V. Furthermore, companies haven taken interest in implementing CHERI-RISC-V processors as commercial products. In 2023, Codasip announced the development of an in-order CHERI-RISC-V processor [25].

There exist multiple instantiations of the abstract CHERI protection model for RISC-V. The University of Cambridge has a series of technical reports[2] that document the development of the abstract CHERI model as well as versions of the CHERI-RISC-V ISA. The most recent publication is the *CHERI Instruction-Set Architecture v9* [93]. Beyond that, there exists the *CHERIoT* ISA, which was originally developed by Microsoft [2, 3]. This ISA is focused on embedded systems and does not offer features needed in full-scale application cores. The latest instantiation of CHERI on RISC-V is the draft[3] for standardisation at RISC-V International. In this thesis, all my implementations are focused on CHERI v9 because all stable and openly available versions of the hardware and software stack are on this version.

In its long existence, the CHERI project has seen multiple sub projects of which all are important for an overall successfully running CHERI system. For example, on the architectural and microarchitectural side, this includes the development of capability compression [103] or handling of tagged memory [51]. Both of them are crucial to reach acceptable performance overheads when implementing CHERI systems. On the software side, important building blocks are the CheriABI [28] or the compiler toolchain [78] among many other pieces. The work presented in this thesis builds on major previous work on CHERI systems and in most cases uses this work instead of modifying. In case of modifications, I clearly lay out where the divergence occurs.

## 2.10 Transient execution on CHERI hardware

This section describes the initial work undertaken on transient-execution attacks on a CHERI microarchitecture. In previous work, I evaluated the feasibility of the four Spectre-like attacks, Meltdown-US, and Meltdown-GP on the CHERI-Toooba implementation [34, 35, 104]. The findings were that each of the four Spectre attacks was feasible on CHERI-Toooba, but none of the two previously known Meltdown attacks. In particular, the previous work found Spectre-BTB and Spectre-RSB to be a great threat to the CHERI protection model. The CHERI-Toooba implementation stores capabilities in its microarchitectural

[2] `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-table.html`.
[3] `https://github.com/riscv/riscv-cheri`.

speculation structures and speculative instructions can make extensive use of them. This allows attackers to jump to other protection domains, e.g., the kernel, access data with a kernel capability, and execute kernel gadgets. However, the previous work found that CHERI can limit the privileges of an attacker to a great extent, e.g., the Spectre-PHT attack can only speculate within the capability bounds as otherwise dereferencing the capability would lead to a CHERI exception.

A major contribution of this work is their discovery of a new subclass of Meltdown attacks. Due to raising exceptions late in the pipeline of CHERI-Toooba, an attacker can transiently violate the CHERI protection model. In particular, an attacker can use `CBuildCap` to transiently create a capability out of thin air, transiently widen the bounds via `CSetBounds`[4], and use `CInvoke` and `CUnseal` to transiently retrieve a dereferencable capability from a sealed capability. All attacks build a subclass of Meltdown attacks that has been dubbed Meltdown-CF (Capability Forgery). I revisit these vulnerabilities in Chapter 4 with a full-scale Meltdown-CF transient-execution attack being presented in Section 4.6.

## 2.11 CHERI-Toooba

This section describes the core I am evaluating my work on. CHERI-Toooba is an out-of-order superscalar RISC-V core that has been extended for CHERI-RISC-V [82, 83]. Originally, the core is based on *RiscyOO* written in the Bluespec SystemVerilog Hardware Description Language (HDL) [110]. Toooba is an extension of the *RiscyOO* core by Bluespec Inc, which added support for compressed instructions as well as a debug unit. Apart from extending Toooba to support CHERI-RISC-V, researchers at the University of Cambridge have added various improvements, e.g., a hardware prefetching unit [104]. CHERI-Toooba implements the RV64ACDFIMSUxCHERI architecture, where `xCHERI` is the CHERI-RISC-V extension and everything else denotes standard RISC-V extensions.

The *RiscyOO* core is highly parameterisable and thus is CHERI-Toooba. I have listed my parameter choices in Table 2.1. Whenever differing parameters were used, I have specifically laid that out in later chapters.

CHERI-Toooba does not have a instruction prefetcher. In early experiments, it was found that the instruction prefetcher in CHERI-Toooba did not add notable performance improvements. Table 2.1 does not list L2 TLB information. CHERI-Toooba employs two L2 TLBs: a set-associative TLB with 1024 entries for 4KB pages and a smaller fully-associative TLB for bigger page sizes. The DRAM latency of CHERI-Toooba depends on the Field Programmable Gate Array (FPGA) board it runs on. In order to be able to compare results between multiple boards, the CHERI team implements DRAM delay modules. On

[4] I also discovered violations via `CSetBoundsImmediate` and `CSetBoundsExact`, which are variations of the `CSetBounds` functionality.

| Microarchitectural property | Parameter choice |
|---|---|
| Pipeline width | 2 |
| Load-to-use latency | 4 cycles |
| L1 I/D cache | 32 KiB, 4-way associative |
| L2 cache latency | 21 cycles |
| L2 cache | 1 MiB, 16-way associative |
| L1 I/D TLB | 32 entries, fully-associative |
| Virtual addresses | 39 bits |
| DRAM latency | VCU118: 16 cycles, DE10Pro: 30 cycles |
| Branch direction predictor | Tournament, 1024 entries |
| BTB | 2048 entries |
| RSB | 16 entries |
| STL predictor | 128 entries |
| Data prefetcher | Stride, into L1 D cache |
| Reservation stations | 16 entries each |
| Load queue | 24 entries |
| Store queue | 14 entries |
| Store buffer | 4 entries |
| Reorder buffer | 64 entries |

Table 2.1: Parameters chosen for CHERI-Toooba.

the VCU-118[5] and the DE10 Pro board[6], the DRAM latency of CHERI-Toooba is 16 cycles and 30 cycles, respectively.

CHERI-Toooba is a highly parameterisable core and can be configured to get closer to how an ASIC core would look like. My experiments took place on FPGA boards and are therefore limited in how large the core can be configured. The dual-core CHERI-Toooba setup used in my experiments made full use of many resources on either the VCU118 or the DE10Pro board. This approach is seemingly as close as one can get to an ASIC when conducing academic research. During the evaluation of my experiments, I will comment how I believe my results can be mapped to ASICs.

## 2.12 CHERI-Toooba System-on-Chip

In order to run CHERI-Toooba, many more components are needed as well as an entire toolchain flow. Results in this work are obtained from two platforms: the VCU-118 board and the DE10Pro board. The CHERI team at the University of Cambridge hosts two repositories that build the respective bitfiles for both the VCU-118 board[7] as well as the

[5] `https://www.xilinx.com/products/boards-and-kits/vcu118.html`.
[6] `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=Engilsh&No=1144`.
[7] GitHub repository: `https://github.com/CTSRD-CHERI/BESSPIN-GFE`.

DE10Pro board[8]. The respective repositories connect the CHERI-Toooba CPU and the devices it needs, e.g., AXI devices for accessing memory.

## 2.13 Morello

The Morello project is a collaboration between Arm Limited, the Universities of Cambridge and Edinburgh, SRI International, and Linaro [7, 41]. The purpose of the project is to produce a research vehicle for CHERI on an industry-scale implementation based on the Arm architecture. *Morello* is a term that is often used wrongly and can refer to multiple different things within the academia-industry project around CHERI. In this section, I explain the different parts of the Morello project and the main artefacts produced.

The *Morello ISA* or *Morello architecture* is a CHERI extension of the Arm-v8.2-A architecture [6]. Like other CHERI ISAs, the Morello architecture maps out the abstract CHERI protection model into a concrete architecture. Throughout the whole project, Arm has been clear that this is an experimental extension with any backwards-comparability breaking changes possible at any stage. This is to enable groundbreaking research that might change how the Morello architecture functions. The Morello-ISA has multiple architectural mechanisms that enable the software implementation of fine-grained compartmentalisation on CHERI. However, evaluation of these features is currently in progress, e.g., through library compartmentalisation [38]. This thesis both develops and starts evaluating primitives for fine-grained CHERI compartmentalisation and compares them to the Morello architecture.

The *Morello-CPU* is the ASIC implementation of the Morello architecture. This implementation is based on the Neoverse N1 processor by Arm [9] and taped out on the TSMC N7 process [41]. The Morello-CPU is configured as a four-core SoC with each core being able to be clocked up to a frequency of 2.5GHz. As part of the wider Morello project, Arm distributed hundreds of Morello boxes, with each board being an entire motherboard with the Morello SoC in it.

Besides the hardware implementation, the Morello project includes many pieces of software from various contributors, e.g., Arm's Morello Linux[9] or CheriBSD[10] provided by the University of Cambridge and SRI International [63]. Software on Morello is still an ongoing research project and regularly sees new pieces of software being ported to run on the Morello CPU, e.g., approximately 10,000 third-party software packages have been compiled for the pure-capability Application Binary Interface (ABI) that accesses memory only through CHERI capabilities [91].

[8] GitHub repository: `https://github.com/CTSRD-CHERI/DE10Pro-cheri-bgas`.
[9] `https://linux.morello-project.org/docs/`.
[10] `https://www.cheribsd.org/`.

## 2.14 Summary

This chapter describes the current state of ISAs and microarchitectures leading to transient-execution attacks. These attacks can have multiple root causes, but have in common that a processor speculatively accesses secret information, which can be leaked via side-channel attacks. Next, this chapter describes defence mechanisms and hardware-software contracts, which are an early attempt to argue about sensible speculation possibilities in microarchitectures. Furthermore, this chapter gives background information on CHERI and compartmentalisation, which this thesis builds on in later chapters. Last, this chapter describes CHERI hardware with a focus on CHERI-Toooba, which was used for previous work reproducing transient-execution attacks on CHERI.

# Chapter 3

# Architectural contracts for safe speculation

The microarchitecture of a processor should enforce the memory-protection mechanisms of its ISA specification. Threat models for such processors assume an attacker who can execute arbitrary instructions, but is nonetheless expected to be entirely constrained to reading and writing memory mapped in its address space and accessible in its ring. Threat models for general-purpose cores have typically excluded side-channel attacks. For example, AMD's threat model for secure encrypted virtualisation [52] states that "with standard software security practices, code which is sensitive to such side channel attacks... should be written in a way which helps prevent such attacks." While side-channel attacks by themselves can be tricky to undertake, when combined with a transient-execution attack, secret data can be read out at KiBs per second [58]. As a consequence industry has seen such attacks as an unacceptable risk.

Unfortunately, combining side-channel attacks with transient-execution attacks has enabled attackers to read arbitrary memory locations in violation of ISA-defined protection [58, 64]. These attacks have not been viewed as acceptable by the industry, prompting both software and hardware responses. Clearly the threat model for processor vendors has shifted to include these transient-execution attacks [11].

Throughout this chapter and Chapter 4, I therefore assume a threat model where an attacker has arbitrary code execution and can observe timing information, e.g., through the cache hierarchy, and other side channels to exfiltrate information. Arbitrary code execution is reasonable as many scenarios allow attackers to inject their code directly, e.g., JavaScript programs or the Berkeley Packet Filter (BPF). I assume that the victim is not collaborating with the attacker, as in the case of a covert channel. In case of collaboration, easier attack scenarios, e.g., pure covert-channel attacks, would suffice in order to leak a secret accessible to the collaborating process. I also assume that the victim code is written to be free of secret-dependent memory access patterns, as is the industry-accepted expectation for cryptographic code [52]. Given these restrictions, I

do not expect any secret of the victim to be observable to the attacker. No amount of observation of failed speculation in the attacker can reveal a secret from the victim.

The work presented in this chapter is based on my publication that was presented at ICCD 2023 [37].

## 3.1 Architectural contracts

Transient-execution attacks can be divided into three phases:

1. Speculative access to a secret value.
2. Transmitting of the secret value through a side channel.
3. Retrieving the secret through a side channel.

If any of three phases is unsuccessful, the entire attack will be unsuccessful.

In order to approach transient-execution attacks, previous research has come up with speculation contracts. A contract is a measure in the ISA that provides guarantees about state that is traditionally not part of the ISA. In my case, this is speculative execution. Traditionally, architectures would not argue about speculative execution because it is a microarchitectural trait. However, transient-execution attacks make it necessary to have guarantees in the ISA, which implementations need to provide and software can rely on. More background information is presented in Section 2.6.

My contracts focus on controlling speculative execution rather than side-channel prevention. This approach is both stronger and more practical than architecturally restricting side channels. Restricting speculative execution is stronger because it prunes the data that can be accessed in the microarchitecture before it can be leaked through side channels. Restricting speculative execution is more practical, because side channels are notoriously difficult to enumerate, and even known side channels are difficult to define in terms of the ISA. Attempting to close all side channel leaves open that an attacker can come up with a different – not yet known – side-channel and thus still has a way of leaking the secret. However, when one prevents the access to a secret in speculation, it is irrelevant which side-channel an attacker can construct because no secret value will exist in the first place.

My contracts are placed on the architectural level. This mandates that the microarchitecture implements these contracts and can be tested against them. For software, the contracts depict security primitives that can be assembled to construct secure programs. Via formal verification, programs then can be declared to not be vulnerable against a certain class of transient-execution attacks. This is depicted in Figure 3.1.

In this thesis, I will focus on constructing architectural contracts and reason about their definition (see Sections 3.5, 3.6, and 3.7). Furthermore, I test the CHERI-Toooba microarchitecture against my contracts in Section 3.9. The verification of software mitigation

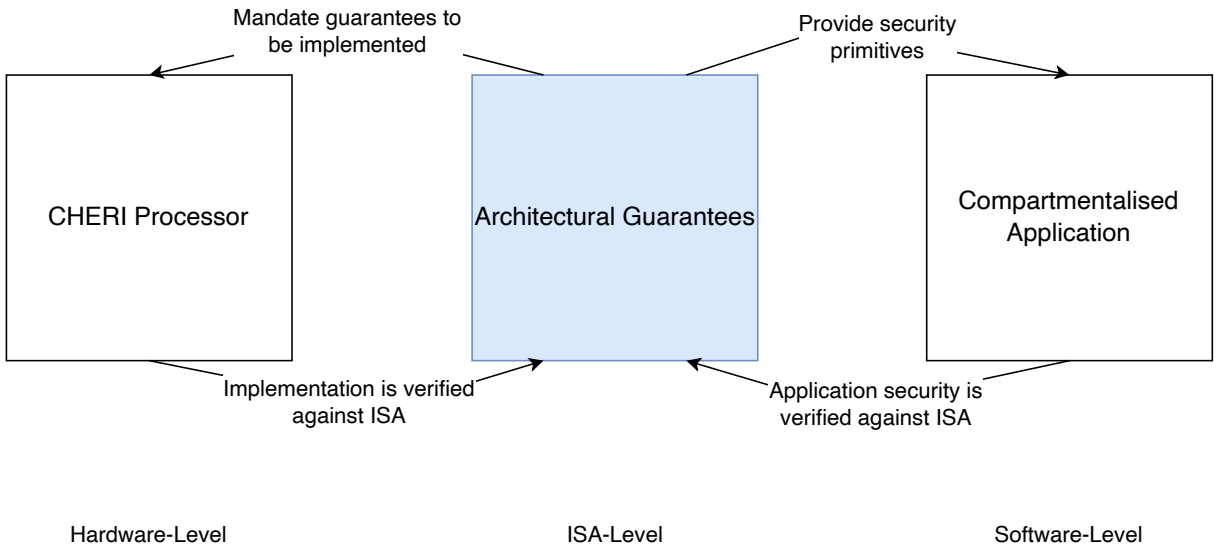



Figure 3.1: The function speculation contracts take on the overall system perspective.

mechanisms against transient-execution attacks is out of scope for this thesis and remains future work.

## 3.2 Common vocabulary

My contracts are presented in prose for accessibility. In order to to avoid ambiguity, I define the use of the following terms:

A *compartment* is a unit of software decomposition expressed architecturally to be enforced by hardware. Examples might be privilege level (expressed in rings) or address spaces (expressed in process IDs or page table roots). More fine-grained compartments are subject to research presented in Chapters 6 and 7. Many systems might do compartmentalisation on a process-level, where each process becomes its own compartment as well as making the kernel its own compartment. When doing a compartment switch, e.g., a system call changing from a user space compartment to the kernel compartment, the microarchitecture needs to be made aware of the compartment change.

I define *the core* as reaching from the execution pipeline to the load/store queue, but not to the L1 caches. This model assumes unused bytes of cache lines cannot leak, and that the cache itself, including any prefetching, does not perform data-dependent operations sufficient to create side-channels. In this research, I also ignore asynchronous agents like the page-table walker and Direct Memory Access (DMA) engines.

An instruction is *fetched* if it is in the front-end of the processor. The front-end is characteristically in-order and flushed at lower cost than the out-of-order back-end. Instructions that are merely fetched need not modify predictor state.

For my contracts the difference between an instruction being *fetched* and *executed* is essential. A fetched instruction does not modify predictor state or access data caches. An

instruction being executed can do both and potentially also produces a new data value in the *core* when data-dependent action is taken. This definition and the ones below clarify the use of words in the contracts, but they do not constrain microarchitectures. This is solely done by the contracts themselves.

A *branch* is an instruction that affects control flow. This includes conditional branches to static targets and jumps to dynamic addresses both within and across functions (also referred to as *calls*), but not exceptions or interrupts.

A memory operation is *issued* when its address is translated. A memory operation is *executed* when it accesses caches.

An event is *branching-non-speculative* if all previous branches have been executed.

An event is *transient* if it occurred speculatively but did not commit. A branching-non-speculative event may still be transient.

## 3.3 Architectural independence of contracts

The contracts presented in the following sections are architecture-independent and can be mapped to any ISA. The contracts argue about general architectural principles not specific to any particular ISA. In the following sections, I perform the step of mapping the contracts to RISC-V, and give examples of which scenarios are allowed and disallowed. It is future research to map my contracts onto other architectures.

## 3.4 Design requirements of contracts

I designed my contracts carefully from the following observations:

- Limiting speculation freedom in modern microarchitectures negatively impacts performance.
- Egregious transient-execution attacks rely on shared microarchitectural state.
- Software mitigation mechanisms for transient-execution attacks rely on assumptions about microarchitectures, not on guarantees.

The first two observations were made in previous work [34, 35]. The last one, comes from looking at common software mitigation mechanisms like Speculative Load Hardening [23] or pointer masking [72]. Based on that, I have developed my contracts to satisfy the following constraints:

- Small single-digit performance impact on out-of-order microarchitectures.
- Provide building blocks to mitigate egregious transient-execution attacks.

My contracts are not to be seen as a complete mitigation mechanism for transient-execution attacks. Rather, they are a necessary guarantee, which is used to by software to build secure execution. For future work on this, please see Section 8.3.

## 3.5 Translation speculation contract

The Translation Speculation Contract (TSC) guarantees that architectural memory translation and page permissions are enforced continuously in the microarchitecture, including all speculative execution. This ensures that no unmapped memory access is issued from the core. Stated precisely:

*Translation Speculation Contract* (TSC)

All instruction and data-memory operations executed in speculation must be mapped in the page table, and allowed by current page permissions.

The Meltdown vulnerability [64], which can be attributed to a lack of specification of the limits of speculation, violates TSC, and was considered an egregious violation of programmers' expectations. Meltdown has not been considered endemic of high-performance microarchitectures; many high-performance implementations do not share this vulnerability and newer implementations avoid it without performance penalty [16]. Therefore, I conclude that the Meltdown vulnerability could be attributed purely to a lack of this architectural speculation contract.

## 3.6 Branching speculation contract

Control flow is the biggest challenge for a complete definition of speculative behaviour. Instruction Fetch is the earliest stage of the pipeline and must operate with minimal knowledge of the instructions that are ostensibly directing it. Branch prediction is therefore traditionally the most speculative aspect of microarchitecture.

In order to define useful constraints for speculative control flow, I strategically limit which instructions will be *executed* rather than merely *fetched*. Side-channel attacks from speculative execution are useful when they can exfiltrate an illegal value; the production of an illegal value generally happens in the Execute stage of the pipeline, e.g., when a load is actually performed. This allows the front-end of the pipeline to speculate freely until decoding the instruction stream. Allowing wild instruction fetch assumes that it is

not possible to leak the contents of memory through the front-end of the pipeline. This should at least require constant-time decoding. Without constant time decoding, e.g., for micro operations [77], timing information on which instructions are executed will leak. After Instruction Decode, the microarchitecture can intelligently enforce architectural speculation contracts with knowledge of the instructions that would be executed.

*Branching Speculation Contract* (BSC)

1. Direct jumps (unconditional with immediate target) will be followed in speculative execution only by the jump target.
2. Direct branches (conditional with immediate target) will be followed in speculative execution only by one of the two possible targets.
3. Instructions that cause exceptions will be followed in speculative execution only by the non-faulting path.
4. Indirect branches (register target) will be followed in speculative execution only by previous architectural indirect-branch targets from the current compartment.
5. Returns will be followed in speculative execution only by the instruction after a previous call from the current compartment.[a]

[a] This does not guarantee that a return will speculate to the correct return address.

The Branching Speculation Contract (BSC) is designed to allow reasoning about the possibilities of speculative execution within a compartment, and also to provide guarantees between compartments.

BSC **item 1** only allows execution to follow the correct target of a direct jump. In this case the target is known in Decode and mispredictions can be flushed before execution. This allows two or more basic blocks to be executed as if there was no control-flow edge between them, and they were a single combined block. For example:

```
labelA:
    jal x0, labelB // Jump to labelB by adding an offset to
    ...            // the current PC
labelB:
    ...
```

The direct jump can proceed only to `labelB` – no other speculatively executed path is allowed. This explicitly forbids straight-line speculation, which would be the instruction following the `JAL` instruction.

**Item 2** precludes the processor from conducting wild target speculation for direct branches. For example:

```
    blt x6, x7, labelA // May only go to two destinations:
    ...                // The next instruction or labelA
labelA:
    ...
```

After the conditional branch, speculative execution can proceed only to two instructions: either to the next instruction, or to the instruction at `labelA`. This constraint is also foundational for current Spectre mitigations, which assume that control flow at a direct branch might take the wrong path due to a misprediction in the PHT, but will not select an arbitrary target from the BTB [23, 111]. Direct branches are nevertheless likely to use a simple BTB for all early branch predictions, but microarchitectures should not proceed to execution with a conditional branch prediction that is not one of the two allowed targets. A branch discovered in Decode that is not followed by one of the two valid targets (determined entirely from the PC and an immediate value) will cause a flush of the Instruction Fetch pipeline. Redirecting from Decode is common and found in commercial implementations, e.g., in the AMD's Zen architecture [24]. As a result, any Execute pipeline will see only instructions from architecturally reachable basic blocks.

**Item 3** guarantees that exception paths will never be speculatively initiated by the processor, and that the straightforward non-faulting path will always be taken in speculative execution. From a microarchitectural point of view, every instruction that potentially throws an exception can be seen as a branch. The instruction can take two paths: The next instruction in program order or the first instruction of the exception handler. While it could be tempting to speculate on exceptions such that repeated exceptions will train the processor to speculatively enter the exception handler, I recommend this simpler constraint. Exceptions are not a common case, and mixing instructions of different privilege levels in speculative execution increases the reasoning burden for software. Obeying this constraint in the contract is not only essential to provide security, but also enforces clean ISA design. There is no incentive to optimise a microarchitecture, e.g., by doing prediction on exception throwing instructions because exceptions are supposed to be the rare case instead of the normal case.

With **item 4**, only previously committed targets from this *compartment* may be predicted, allowing reasoning about the complete set of potential speculative targets of an indirect branch. Predicting targets learned from transient execution would make static analysis impossible. In fact, I find in Section 3.11 that excluding transient targets in the BTB can improve performance. Item 4 also prevents indirect branches from defaulting to straight-line prediction, which has been a source of transient-execution attacks [15]. As the instruction after an indirect branch is generally not a previous target, straight-line predictions are disallowed unless it has been a previous architectural branch target.

The compartment constraint of BSC item 4 allows software to reason about who can influence indirect branch speculation without unduly restricting the implementation.

This constraint is already enforced for ring compartmentalisation on recent commercial processors, e.g., on modern Arm implementations [10], to prevent the Spectre-BTB attack that trains kernel BTB targets from userspace. This constraint should now be specified in the architecture to guarantee this property for all implementations.

**Item 5** describes standard call stack prediction using the RSB, but allows training only from branching-non-speculative function calls. This is expected behaviour and thus does not unduly restrict microarchitectures. However, implementing item 5 might lead to performance penalties. Once scenario might be calling many short functions, all of which follow an unresolved direct branch. This can lead to performance reduction because the RSB has not yet been populated. The results presented in Section 3.11 show that delayed pushing to the RSB brings an overhead. While this overhead is not large, it gives reason to optimise the contract or the implementation. This principle in item 5 forms the basis of the broadly deployed Retpoline Spectre-BTB mitigation recommended by Intel [48]. Nevertheless, this expectation is not specified architecturally; both Intel and AMD processors use BTB predictions for the call stack in certain cases, allowing Retpoline mitigations to be bypassed [100].

Constraining speculation training to branching-non-speculative jumps from the same compartment, as specified in items 4 and 5, enables full analysis of possible speculative paths in a compartment, and also guarantees non-interference between compartments. Speculation in a compartment can only be directed by branching-non-speculative execution in that compartment. This makes it possible to enumerate a complete set of previous branch targets that can be re-used for future speculation, regardless of the varying degrees of history allowed by microarchitectural space constraints. This style of constraint also appears practical, as high-performance commercial processors are shipping now with this behaviour for ring-based compartments.

While having BSC items 4 and 5 allows us to reason about the speculation targets, it does not disallow misspeculation and nor does it cut off that a compartment can impact the speculation in another compartment. For example, a compartment might be able to influence the data of another compartment, which then can lead to control-flow divergence.

```
// unsanitised value from attacker in a0
slli  a0, a0,  2 // Align to four byte address
auipc t0,  0     // Get current pc and store it in t0
add   t0, t0, a0 // Perform pointer arithmetic
jalr  ra, t0     // Jump to address calculated above
```

Listing 3.1: Attacker-controlled pointer arithmetic.

In Listing 3.1, pointer arithmetic is done on a value that is directly controlled by the attacker. This allows the attacker to train the BTB entry behind the `JALR` instruction. While this scenario is allowed by our contract, software can take action to avoid this, e.g., by sanitising every value that is not controlled by its own compartment.

## 3.7 Data-value speculation

Data-value speculation is used in modern processors in various forms. Memory is the most important target for data-value speculation, and speculatively reordering loads and stores is a basic feature of modern out-of-order cores. For this initial work, I present the Memory Ordering Speculation Contract (MOSC) as a complete data-value speculation contract for memory; no other form of data-value speculation for loads is allowed.

*Memory Ordering Speculation Contract* (MOSC)

Speculatively executed loads will return a value that has been held at that address while in this compartment. Speculatively stored values may be loaded from this compartment; only committed (non-transient) values may be loaded from other compartments.

These semantics of MOSC are a compromise between architectural semantics and microarchitectural freedom. MOSC allows the most natural form of speculative memory reordering without compromising compartmentalisation security guarantees as long as software is careful about memory initialisation.

MOSC allows old values of a memory location to be visible in transient execution, but prevents more esoteric cases where the value of the memory location might be predicted before acquiring a legitimate copy from memory.

Data-value speculation is difficult to specify in general without removing all constraints on the possibilities of transient execution. I recommend that architects compose an architectural contract for each category of data-value speculation to facilitate clear thinking about ramifications for speculative execution attacks and also a reasoned microarchitectural design.

## 3.8 Security evaluation

Architectural speculation contracts embrace an ISA threat model that includes transient-execution attacks in the effort to improve architecture-wide security guarantees. In this section, I evaluate the security guarantees of my proposed contracts in the face of transient-execution attacks. I follow the classification of attacks from Canella et al. [22].

### 3.8.1 Spectre defence

Spectre-like attacks are classified based on the prediction mechanism they exploit: Spectre-PHT, Spectre-BTB, Spectre-RSB, and Spectre-STL.

Whereas BSC limits direct branch-direction prediction to the two architectural possibilities via constraint 2, it does not natively prevent Spectre-PHT attacks that use a transient wrong path to bypass security checks. Note, however, that pointer masking [72], a software mitigation for Spectre-PHT for out-of-bounds accesses, relies on BSC constraint 2 for guaranteed mitigation. While Spectre-PHT is not natively prevented by BSC, this contract ensures that both Speculative Load Hardening [23] and pointer masking [72] are effective mitigation mechanisms against Spectre-PHT

BSC limits the speculative targets of indirect jumps, calls, and returns to committed targets from the same compartment as a defence against Spectre-BTB and Spectre-RSB. Attackers can only train the BTB and RSB with architectural targets, so cannot redirect control flow to arbitrary targets. Furthermore, BSC does not allow an attacker to train their own targets into another compartment. However, BSC constraints 4 and 5 do not forbid wrong target speculation as long as a jump to that target has been committed previously in this compartment. While an attacker cannot inject their own targets, an attacker might still mistrain another compartment, e.g., through BTB conflicts, to misspeculate to local targets that may reveal secrets or through pointer arithmetic as depicted in Listing 3.1. Nevertheless, BSC does enable a compartment to enumerate all potential speculative targets of an indirect branch, by analysing only its own code.

Note that my contracts argue about the targets, but they do not argue about the temporal ordering of these targets. One example of that is a Spectre attack, which manages to jump out of a cryptographic function before the secret could be sufficiently encrypted [86]. In this case the jump target is correct, but it was used too early in the speculation flow. In Chapter 7, I develop a mechanism to counter such attacks.

Spectre-STL attacks mistrain the memory disambiguation predictor, with the goal of transiently loading secret data to be exfiltrated via a side-channel attack. MOSC prevents inter-compartment Spectre-STL attacks if secrets are cleared before memory is shared between domains.

As summarised in Table 3.1, my contracts limit the inter-compartment attack surface for Spectre attacks.

| | Intra-Compartment | Inter-Compartment |
|---|---|---|
| Spectre-PHT | ✗ | ✗ |
| Spectre-BTB | ✓(limited) | ✓ |
| Spectre-RSB | ✓(limited) | ✓ |
| Spectre-STL | ✗ | ✓ |

Table 3.1: Spectre security evaluation for intra-compartment and inter-compartment attacks mitigated by BSC.

| | Defence |
|---|---|
| Meltdown-US | ✓ |
| Foreshadow | ✓ |
| ISA-dependent | ✗ |

Table 3.2: Meltdown security evaluation.

### 3.8.2 Meltdown defence

Meltdown-US illegally reads secrets from kernel pages that are still mapped in the page table – despite permissions prohibiting the access. TSC is an architectural formalisation of hardware Meltdown-US protections and fully mitigates Meltdown-US attacks. TSC also prevents Foreshadow [19], a variant of Meltdown targeting Intel SGX encrypted pages.

Many ISA-specific Meltdown-like attacks have also been developed [22], e.g., Meltdown-NM [88] for x86 floating point. Most of those attacks are specific to one ISA and particular specific subsets of it. My contracts are instruction-set architecture independent so do not consider such ISA-specific attacks. Including ISA-dependent contracts in my set would pollute the architectural interface. ISA-dependent threat vectors will need their respective speculation contracts. One example of that are speculation contracts for CHERI, which are presented in Chapter 4. The results are summarised in Table 3.2.

### 3.8.3 Microarchitectural data sampling defence

MDS attacks harvest secrets by exfiltrating undefined data returned in the microarchitecture during failed speculation, e.g., the Fallout [21] attacks targeting the store buffer or RIDL [84] targeting line fill buffers. Both TSC and MOSC prevent MDS attacks. TSC prevents MDS attacks against data forbidden by memory translation or permissions. MOSC also prevents MDS attacks by excluding all data-value speculation (including undefined data) besides memory reordering, which must return data belonging to the same address and compartment.

### 3.8.4 Closing unknown vulnerabilities

A key goal of specifying reasonable semantics of speculative execution in the architecture is to avoid vulnerabilities in the design phase of new processors. Addressing vulnerabilities in shipping implementations is unacceptable on processor-design time scales. Architectural speculation contracts force thorough consideration of speculative behaviour in the processor design phase.

My contracts and their implementation in CHERI-Toooba solved two vulnerabilities that were discovered after I had completed the implementation.

The first vulnerability was Straight-line Speculation [15]. BSC constraint 4 requires that indirect branches must be followed in speculative execution by previous indirect branch targets. We as a research group discovered that CHERI-Toooba would speculatively execute the instructions immediately following an indirect branch if no target was found in the BTB, violating BSC. This possibility was removed in the implementation in order to obey the contract and because it was found to cause a performance issue in one case. We later discovered that straight-line prediction was a source of exploitable vulnerabilities in commercial processors.

The second vulnerability (which I unknowingly avoided) was Retbleed [100]; it leveraged cases where the BTB is used to predict return addresses to bypass Spectre-RSB defences. BSC constraint 5 states that function returns must arrive at a previous call site, and CHERI-Toooba was tested to conform with this behaviour. Retbleed showed that Intel and AMD processors sometimes resort to BTB predictions for returns, allowing bypass of Retpoline defences. Retpoline was proposed by Intel itself in 2018 as a Spectre-BTB mitigation [48]; if the assumed behaviour had been encoded in the architecture and tested on new implementations, several generations of vulnerable implementations could have been avoided.

## 3.9 Testing speculative contracts

As with all components of an ISA, architectural speculation contracts should be verifiable. I developed directed-random testing engines for each of the proposed speculation contracts.

The IntroSpectre verification effort shows the difficulty faced by the current state of the art [40]; IntroSpectre uses debug tracing in simulation and directed-random test generation to discover leakage of state between privilege levels. However, the IntroSpectre approach is both inconclusive and non-portable. Inconclusive because a lack of specified behaviour means that a violation does not necessarily imply an error, but merely a sequence to be investigated. Non-portable because events must be detected using implementation-specific tracing as they could not assert clear architectural invariants.

I intend to provide a clear set of contracts against which to test, and a portable testing strategy. For a portable testing approach, I relied on the RISC-V Formal Interface (RVFI) [102] a standard architectural tracing format also used by other verification efforts. Furthermore, I used the Hardware Performance Monitoring (HPM) extension to detect microarchitectural events; HPM counters provide a portable bridge between architectural state and the speculative behaviour guaranteed by my contracts. Whereas arranging scenarios where counters reliably indicate violations was sometimes complex, this approach is portable to a family of implementations that share the same architectural contracts.

My contracts enforce microarchitectural principles and need to be made architecturally visible. Using a form of measurement already present in the architecture would be an implicit measurement, e.g., instruction timing, and thus be imprecise. Using the HPM framework remains the minimal invasive method to reach architectural testing of my contracts. My approach adheres to the following principles:

1. A clear architecture against which to test.
2. An implementation-agnostic interface to my test engine.
3. The ability to shrink counterexamples.

I built my generators in the TestRIG framework for RISC-V processors [50, 79], which is supported in CHERI-Toooba. In addition, I detect microarchitectural events using the RISC-V standard HPM interface, which is part of the RISC-V privileged specification implemented in recent RISC-V processors [30]. TestRIG also provides automatic test case shrinking to greatly improve the user experience. As shown in the counterexamples in this section, my generators can often reduce a violation to fewer than 5 instructions, enabling engineers to practically analyse a replay in simulation, with full microarchitectural tracing.

My approach to discovering contract violations is similar to Revizor [68], which uses random test generation on x86 CPUs to discover violations of advanced side-channel speculation contracts that correspond to Spectre vulnerabilities, including Spectre-PHT and Spectre-STL, but not considering any Meltdown attacks. Revizor generates x86 programs and samples performance counters to detect microarchitectural events. Revizor's primary drawback is the extensive preparation phase to ensure that a high-performance x86 CPU did not diverge in unexpected ways from its greatly simplified model. A comprehensive architectural contract that is an active design target of the microarchitecture would make Revizor-like efforts more general and useful.

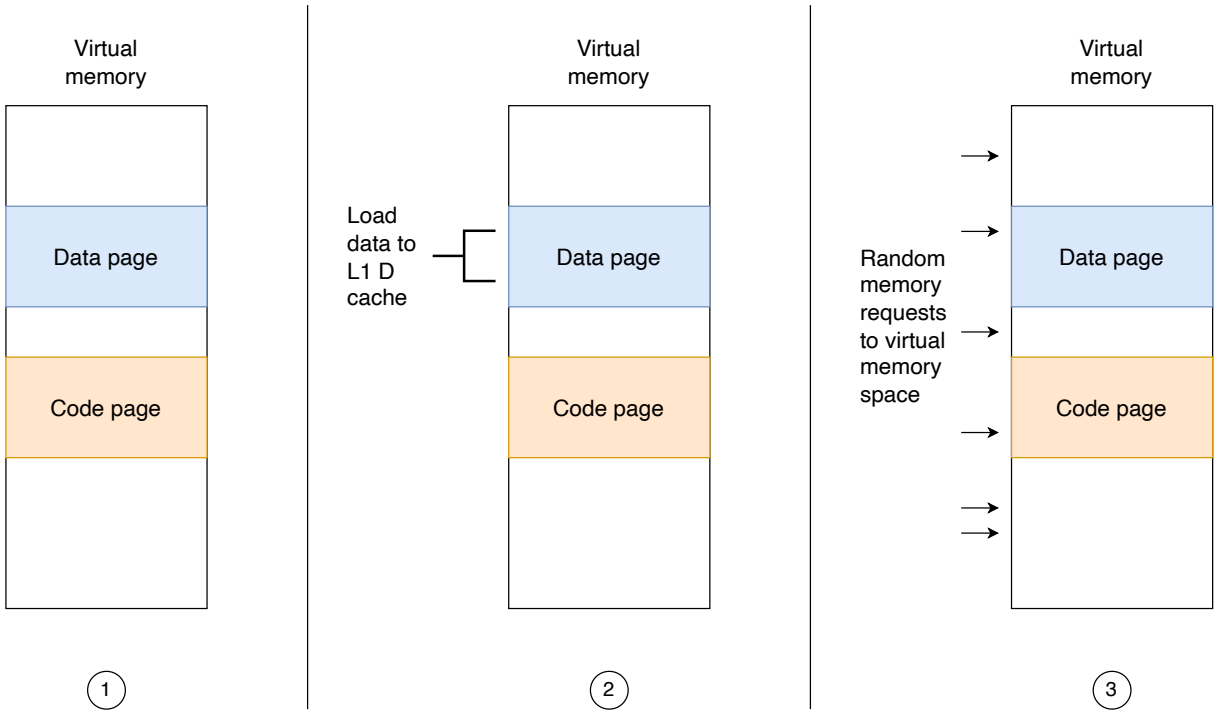



Figure 3.2: Steps when testing CHERI-Toooba against the Translation Speculation Contract. First, two pages are set up in the page table. Second, the data from the data page is loaded in the L1 D cache. Third, the generator fires random memory requests to test whether TSC can be violated.

### 3.9.1 Translation speculation contract testing

TSC requires that transient execution respects page-table mappings and permissions. My TSC generator populates a page table with one code and one data page, and arranges for the data-cache miss counter to register any accesses outside these pages. The code page is set up as executable-only, whereas the data page is readable and writeable. Next, data on the data page are loaded into the L1 D cache. In the last testing phase, random memory instructions then attempt to access disallowed pages. If the L1 D cache miss counter is incremented, a TSC violation has taken place. The testing steps are summarised in Figure 3.2. As expected from the results of previous research [34, 35], my generator did not detect a TSC violation in the CHERI-Toooba core.

At the time of obtaining these results, CHERI-Toooba has not yet employed hardware prefetchers. As described in Chapter 2, CHERI-Toooba now has a data prefetcher, which fetches into the L1 data cache. Prefetching into the L1 data cache does not lead to a TSC violation. However, when hardware prefetchers fetch data in the core itself, they are a likely source of TSC violations.

### 3.9.2 Branching speculation contract testing

BSC consists of five constraints that I test via two generators: *branches and jumps* (constraints 1, 2, 4 and 5), and *exceptions* (constraint 3).

#### Branches and jumps violations

My branch-and-jump test generator seeks to detect violations of BSC constraints 1, 2, 4, and 5. This ensures that direct jumps speculate only to the intended target; direct branches speculate only to one of the two possible targets; indirect jumps speculate only to a previous jump target; and returns speculate only to previous calls. Testing these properties in CHERI-Toooba required implementing specialised counters using a small amount of state.

I detect direct-jump and direct-branch violations using custom assertions with counters in the Execute stage; these assertions register an event if the predicted nextPC is an implausible target.

For indirect jumps, I implemented a searchable vector of previously committed indirect jump targets and count *wild* predictions that are not found in this set. For returns, I similarly record previous call sites and count predicted return targets that are not in this set. The implementation is a concurrently searchable structure that maps PC values to target addresses. While the implementation of this state would be costly on an FPGA or on an ASIC, testing is done in simulation where area is not a constraint.

To prevent overflow of my structures, my generators ensure a limited number of jump targets by creating cycles of instructions with jumps, branches, or targets at fixed offsets. Using the same PC values for different jumps is also a handy strategy to create misspeculation because CHERI-Toooba will not expect a new jump at already seen PC values.

Any branch that does not conform to my expectations for BSC is considered a *wild jump*, and is counted in the wild-jump counter. My generator produced a class of counterexamples that has speculative BTB training in common. The following counterexample is a member of this class:

```
csrci mcountinhibit, 8 // CSR instruction flushes the
                       // pipeline
jalr x1, x10, 21       // Speculative training of BTB
srliw x28, x29, 1      // Additional instruction for
                       // triggering microarchitectural state
```

The Control and Status Register (CSR) Clear Immediate (`CSRCI`) instruction is a system instruction that causes a pipeline flush in CHERI-Toooba. The `JALR` instruction speculatively executes after the `CSRCI` instruction, but is flushed from the pipeline when `CSRCI` commits. Since the first execution of `JALR` does not commit, it does not produce a bonafide

previous architectural target. Nevertheless, the transient `JALR` trained the BTB such that the `SRLIW` target executes speculatively after the flush, violating BSC constraint 4.

A return that violates BSC is considered a *wild return*, and is counted in the wild-return counter. My generator produced one class of counterexamples, which is caused by RSB training in speculation. This class is a superclass of the class responsible for wild jumps.

```
csrrs x21, hpmcounter7, x0 // CSR instruction leading to a
                           // pipeline flush
jalr x1, x0, 0             // Speculative jump to a target
                           // populating the RSB
jalr x0, x1, 0             // ``jalr x0, x1, 0'' is a ``ret''
                           // Return consuming a speculative
                           // target from the RSB
```

The CSR Read-Set (`CSRRS`) instruction flushes the pipeline. However, the two next instructions – a call and a return instruction – have already speculatively executed before the flush. The call has speculatively populated the RSB. Therefore the return – when re-executed – consumes the speculative target and therefore violates BSC constraint 5.

While the examples above seem to be a benign violation of BSC, one can easily think of more dangerous ones, e.g., if the jump instruction is on a misspeculated path and therefore polluting the BTB. This scenario would allow to train the BTB in speculation creating non-architectural targets.

#### Exception violations

BSC requires exceptions to speculate the non-faulting path. My generator produces random non-branching instructions which sometimes cause exceptions. For each instruction, I check in the final stage of the respective execution pipeline whether this instruction will raise an exception and if so whether the predicted next PC has another value than the PC for the non-faulting path. I could not identify any violations of BSC constraint 3 and thus conclude that this constraint is held by CHERI-Toooba.

### 3.9.3 Memory ordering speculation contract testing

MOSC requires that loads only return values that have been previously held in that address. Within one compartment the stores may be speculative, but loads in other compartments can only return architecturally committed values. I introduced a small amount of state in CHERI-Toooba to track past stores, and count a wild load if a load returns an unseen value. The implementation of this small amount of state is using the same primitives as used for BSC violation testing. As before, I carefully design the instruction generator to ensure the microarchitectural structure does not overflow. I initially fill memory with zeros

| Contract | Violation found |
|---|---|
| BSC item 1 & 2 | ✗ |
| BSC item 3 | ✗ |
| BSC item 4 | ✓(Speculative BTB training) |
| BSC item 5 | ✓(Speculative RSB training) |
| TSC | ✗ |
| MOSC | ✗ |

Table 3.3: Overview of the results when testing my architectural speculation contracts on CHERI-Toooba.

so that one can allow either zero or a recorded previous store value. My generator creates random sequences of loads and stores intermixed with instructions from other classes. This generator did not find a counterexample, indicating that CHERI-Toooba upholds MOSC.

### 3.9.4 Testing overview

I tested CHERI-Toooba for all three contracts: TSC, BSC, and MOSC. My generators could produce test cases the found violations in BSC items 4 and 5, but they could not find violations against any other items or contracts. The results are summarised in Table 3.3.

### 3.9.5 Evaluation of test generators

I traced microarchitectural events triggered by each test. For example, I found that 91.56% of the TSC tests triggered a memory exception due to insufficient privileges, and 9.32% of all BSC jump tests generated a BTB misprediction. At least 94.96% of BSC exception tests raised an exception, and these are categorised into front-end, Rename, ALU, or memory exceptions. The results are depicted in Table 3.4.

## 3.10 Speculative contract enforcement

As described in Sections 3.9.1 and 3.9.3, CHERI-Toooba natively enforced TSC and MOSC at full performance. Therefore, I did not need to carry out implementing mitigation mechanisms for these.

Enforcing BSC on CHERI-Toooba required delaying predictor training until the relevant branch was *branching-non-speculative*, i.e., all previous branches have executed. To prevent mistraining predictors during an exception, one would use coarser measures, e.g., such as fences or flushing, that are a secondary concern for common-case performance. In comparison with more direct side-channel prevention techniques such as those facilitated by Speculative Taint Tracking [109], this limitation is quite subtle and can yield a performance

Table 3.4: Microarchitectural event coverage by generator.

| | BSC Jumps | BSC Exceptions | TSC |
|---|---|---|---|
| ALU usage | 99.57% | 99.90% | 99.90% |
| FPU usage | 0.00% | 94.96% | 98.15% |
| MEM usage | 0.00% | 94.21% | 0.00% |
| Val. forward usage | 96.69% | 99.60% | 99.60% |
| Excp front-end | 0.00% | 93.41% | 0.00% |
| Excp Rename | 0.00% | 94.81% | 0.00% |
| Excp ALU-0 | 0.00% | 92.76% | 0.00% |
| Excp ALU-1 | 0.00% | 92.76% | 0.00% |
| Excp MEM | 0.00% | 94.96% | 91.56% |
| Excp Commit | 0.00% | 94.96% | 91.56% |
| Redirect ALU | 30.23% | 0.00% | 0.00% |
| Redirect Commit | 90.40% | 99.80% | 99.80% |
| Dir. Pred. usage | 30.04% | 0.00% | 0.00% |
| BTB Pred. usage | 32.40% | 0.00% | 0.00% |
| BTB Pred. fail | 9.32% | 0.00% | 0.00% |

improvement. Waiting for state to be branching-non-speculative is both high-performance and convenient for this implementation.

Together with Dr Woodruff, I have developed these mitigation mechanisms. The implementation has been carried out by Dr Woodruff with feedback and discussions with me through various iterations.

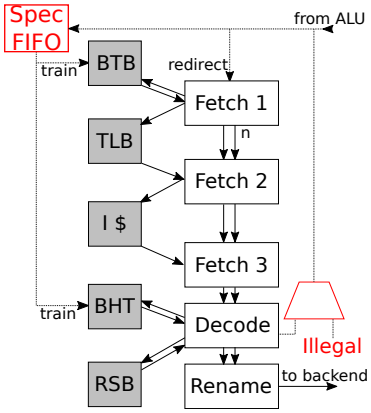
The BSC enforcement in CHERI-Toooba was facilitated by the *SpecFIFO* component. This is a First In First Out (FIFO) structure, which holds data elements as well as speculative state that indicates which in-flight branches this element depends on. The pecularity of a SpecFIFO is that elements in the FIFO are marked as *valid* or *invalid*. Invalid elements are removed from the SpecFIFO. A branch resolution broadcast network updates the validity bits in SpecFIFOs and thus ensures that metadata in SpecFIFOs are updated such that elements belonging to failed speculation immediately disappear from all SpecFIFOs. A speculation state value of 0 indicates that this state is no longer speculative.

**BSC constraints 1, 2, and 3** were already enforced in CHERI-Toooba. A misprediction of a direct jump is detected in Decode when its target is fully known. Direct branches use a dedicated direction predictor[1] in Decode to predict one of only two valid targets. If Decode detects an implausible prediction, it flushes the fetch pipeline and redirects to a legal target.

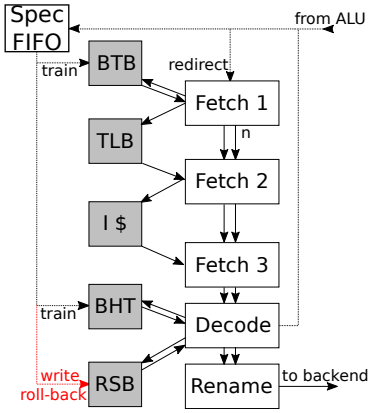
**BSC constraint 4** mandates that all speculative indirect jump targets must have been previously targeted by committed jumps. This requires the BTB to be trained only from *branching-non-speculative* state. To enforce this, we converted the branch training queue into a SpecFIFO. This is a component, which holds a queue of speculative state along with tags that indicate which in-flight branches this state depends on. This branch training

[1]For a more profound explanation of the direction predictor see Section 6.2.3.

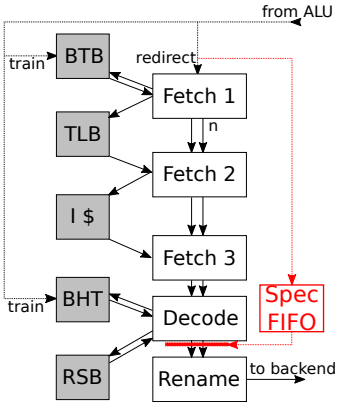
SpecFIFO, shown in Figure 3.3(a), is dequeued only when the head is *branching-non-speculative*. This mechanism guards not only the BTB, but also the direction data in the branch direction predictor. Protecting direction prediction is not essential for BSC, and could have both positive and negative effects on branch accuracy due to delaying training and eliminating training from transient state. Further research needs to show the precise effect of BSC on branch predictor accuracy.




(a) The SpecFIFO component allows training the BTB and BHT with only branching-non-speculative updates. The Decode stage can predict a guaranteed invalid address if no previous target is available. Combined, these help to enforce BSC constraint 4.




(b) The RSB can be rolled back on redirect, preventing desynchronisation caused by speculative execution. In addition, pushing/popping can be deferred until the call/return is branching-non-speculative. These target BSC constraint 5.

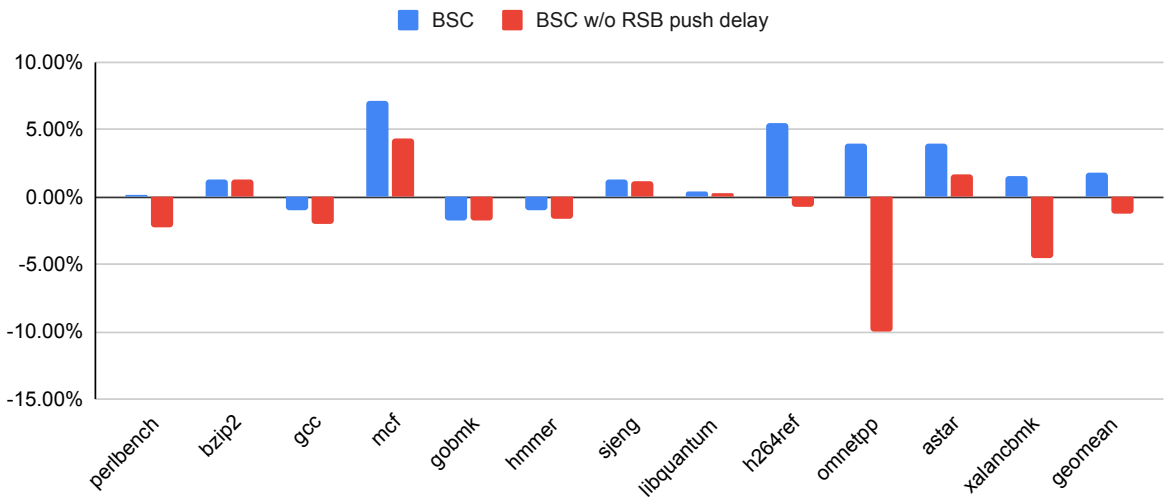



(c) A zero-data SpecFIFO allows instructions to be held at Decode until the instruction that caused the redirection is branching-non-speculative. This is required to enforce BSC constraints 4 and 5.

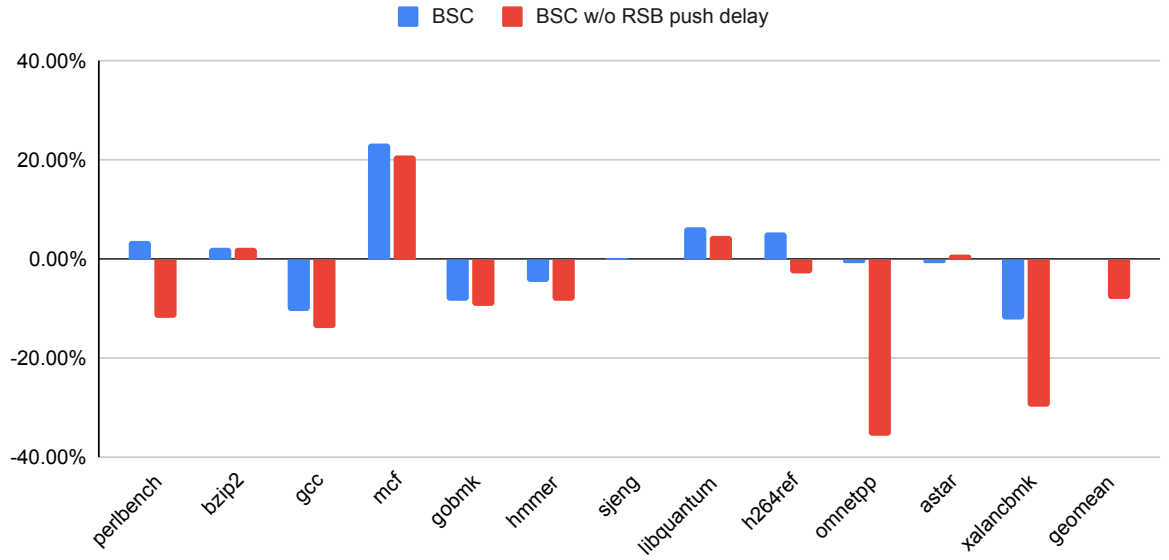
Figure 3.3: Changes made to the CHERI-Toooba front-end in order to enforce the BSC constraints that I found violations for.

As Decode was not able to provide a reasonable target, despite knowing that this was a branch, speculative execution proceeded with the next instruction. This same design pattern was discovered to be an exploitable vulnerability in commercial processor implementations [15]. Straight-line prediction can also lose performance; Dr Woodruff first discovered straight-line prediction in CHERI-Toooba when a cache miss in a load-jump pair in a jump table was followed by a long chain of transient load-jump pairs that also missed the cache. To resolve this, I respond to a missing prediction by pausing Instruction Fetch until I receive an actual redirection from Execute, as depicted in Figure 3.3(a).

**BSC constraint 5** requires that all speculative returns arrive at a previous branching-non-speculative jump target. CHERI-Toooba both pushed and popped the RSB in Decode from the speculative instruction stream. That is, when a *Jump and Link* (`JALR` or `JAL` instructions in RISC-V) was encountered in Decode, PC+4 was pushed to this stack, and when a return was encountered, the top of the stack was popped and predicted as the next PC. This violates BSC constraint 5, as jumps that have pushed to the RSB might be flushed due to misprediction, introducing non-architectural targets to transient




(a) Performance cost and benefit measured in number of cycles.




(b) Performance cost and benefit measured in number of redirections. A redirection constitutes the event where the back-end redirects the control-flow leading to new instructions being fetched.

Figure 3.4: Performance cost and benefit of enforcing the Branching Speculation Contract (BSC) constraints 4 and 5 using the SPECint2006 benchmark. All twelve integer benchmarks were run to completion using the *test* workload.

execution. In addition, performance was lost, as flushed transient instructions could cause the RSB stack to become desynchronised with the actual instruction stream. This can be resolved with two mechanisms, which are illustrated in Figure 3.3(b). First, the RSB stack pointer is rolled back on redirection; this reduced RSB mispredictions by roughly 90% in CoreMark [31]. Second, RSB writes can be delayed in the training SpecFIFO so that only *branching-non-speculative* targets are written to the RSB structure. This is the most expensive aspect of the BSC enforcement, and is therefore listed separately in Figure 3.4.

Separate from training data, redirections themselves also needed protection to enforce BSC constraints 4 and 5. A redirected instruction stream in CHERI-Toooba was able

to reach execution while the instruction that caused the redirection was still speculative. This allows illegal transient targets to execute despite not populating either the BTB or the RSB.

The most straightforward solution would be to store the redirection itself in a SpecFIFO until the mispredicted branch that triggered the redirection had become branching-non-speculative. However, this approach would have increased the misprediction penalty in many cases. Because BSC guarantees only the semantics of speculative *execution*, as opposed to fetch, Instruction Fetch is allowed to proceed with a speculative redirection, but Decode is blocked until the instruction that caused the redirection has become *branching-non-speculative* (see Figure 3.3(c)). This is enforced by a SpecFIFO that holds the speculative state of the last redirection. This mechanism appears to block only very rarely, as performance is hardly affected by this change.

I tested these BSC enforcement mechanisms using the generators described in Section 3.9, which helped identify stray violations and also helped ensure the test routines themselves were flagging appropriate violations. This exercise demonstrated that architectural speculation contracts of this style can be productively tested during development.

## 3.11 Performance evaluation

Figure 3.4 gives the performance overhead of enforcing BSC on SPECint2006 benchmarks [87]. All twelve integer benchmarks were run to completion using the *test* workload. The measurement have been done both with and without the RSB push delay, as this incurs the greatest overhead.

Without the RSB push delay, BSC protections reduce execution time by 1.26% when calculating the geomean. The performance improvement is due to reduced redirections from more accurate predictions, as seen in the lower graph. A redirection is the event that occurs when the back-end *redirects* the front-end to fetching instructions from a new address. These more accurate predictions are due to BSC protections preventing prediction state from being polluted by transient instructions. While transient training sometimes provides positive hints for future behaviour, in these cases they more often hurt performance, and there is a mean 6.9% reduction in redirections without the RSB push delay.

With RSB push delay enabled, there is a geomean 1.75% cycle overhead. This feature increases redirections due to mispredicted short function calls, as explained in Section 3.10.

We were able to enforce the primary constraints of BSC in CHERI-Toooba with near-zero overhead. Given the success of these rudimentary enforcements, I expect this result to generalise to more mature efforts in other microarchitectures as well as in ASIC implementations.

## 3.12 Conclusions

This chapter proposes architectural speculation contracts for instruction sets as a foundation for reasoning about the transient-execution vulnerability of programs. I propose basic contracts, which are designed to be useful for software characterisation, verification, and optimisation, and demonstrate that they are practical for hardware verification. The contracts I have developed resolve the most egregious transient-execution vulnerabilities, and enshrine assumptions required by software mitigations in clear architectural requirements. Therefore, my results support Hypothesis H.1, which claims that conventional processor architectures can be extended with architectural speculation contracts to constrain overall speculation. My research does not conclude whether my contracts are exhaustive, which remains future work.

Despite specifying clear semantics, my contracts avoid limiting performance; the benchmark results for the prototype enforcements in CHERI-Toooba incurred only a 1.75% performance overhead as laid out in Section 3.11. Crucially, I have also demonstrated that these contracts can be verified with microarchitectural tests. Therefore, my results prove Hypothesis H.2, which claims a conventional microarchitecture can be extended to implement the speculation contracts and does not incur significant performance overheads. Given my results, I expect my approach to scale well for ASIC implementations. The RSB push delay solution is a prototype implementation with great room for improvement in future iterations.

Speculation contracts are particularly necessary for CHERI (see Chapter 4), but also in general for security architectural extensions, such as x86 SGX, CET, MPK, and SEV; ARM's MTE and PAC.

I hope this work inspires further exploration of the trade-off between architectural speculation constraints and software mitigations. Stricter contracts require hardware innovation to avoid performance loss, but permissive contracts require potentially expensive software mitigations. Therefore, cross-layer design and optimisation will require expertise from multiple fields, as well as thorough evaluation of each design point. Architectural speculation contracts will be a key tool leveraged by future software countermeasures. These contracts will allow software to provide verifiable transient-vulnerability safety without expensive and conservative software constructs caused by a lack of architectural specification.

# Chapter 4

# Capability speculation contracts

Conventional architectures usually define their security model over two primitives: privilege rings and virtual address spaces. CHERI additionally defines in-memory capabilities equipped with permissions and bounds information, as described in Section 2.9.

In this chapter, I build on the threat model defined in Chapter 3. In addition to rings and address spaces as protection mechanisms, the attacker and the victim are architecturally isolated using CHERI capabilities, preventing explicit reads and writes of code or data. Furthermore, I expect that the attacker and victim code are not collaborating. This holds even if they are in the same address space, e.g., in library compartmentalisation [38]. Therefore, I expect capabilities to be a sufficient measure to ensure protection between different security domains on the same privilege level.

In this chapter, I address this threat model by adding contracts to the instruction-set architecture that constrain capability speculation (in the style of architectural speculation contracts presented in Chapter 3) to allow reasoning about CHERI program safety on all implementations. Part of the work in this chapter is based on one of my publications that was presented at ICCD 2024 [36].

Such capability contracts can be used in two ways:

1. To embody the transient-execution threat model for microarchitecture design. A CHERI microarchitecture can be tested or formally verified against these properties to demonstrate it has sufficiently addressed threats from transient execution.

2. To provide a basic building block that can be used in order to determine whether a piece of software is secure against transient-execution attacks with respect to their higher-level threat model. I envision this property to be useful in software verification.

While the contract in this chapter is shaped around CHERI, its underlying principle of checking privileges before doing accesses can also be applied to other architectures similar

to CHERI. One sample architecture which could have benefitted from speculation contracts is Intel's MPX architecture, which aimed to tackle memory safety violations [69]. However, it is important to note that a specific contract always needs to be tailored to the concrete architecture in order to obtain useful protection properties.

## 4.1 A capability speculation contract

In CHERI, bounds and permissions checks must not only be safe in non-faulting, in-order execution, but also in transient, faulting execution [35, 96]. CHERI invariants include:

- CHERI capabilities are unforgeable; capabilities are derived only from capabilities of greater or equal privilege.
- Memory can be addressed only through a capability describing and authorising access to that address.

The first requirement is naturally enforceable in speculation, as pipelines generally forward values that are legitimately calculated from register state. The second requirement is also naturally enforceable, as capability metadata is bundled with the address and can be verified before issuing requests to memory. These two requirements together give rise to a powerful emergent property I call the Capability Speculation Contract (CSC). I posit CSC is achievable in high-performance microarchitectures, which I demonstrate in Section 4.7.

*Capability Speculation Contract* (CSC)

All instruction and data-memory accesses issued in speculation must be authorised by capabilities:

1. in the committed register file;
2. in memory transitively reachable through 1.

In other words, a CHERI processor should act – even in speculation – only with rights transitively reachable from its architecturally committed register file. CSC does *not* forbid speculation on capabilities, but it does forbid using speculatively manipulated capabilities that cannot be found in the architectural register file and its transitive closure. Furthermore, an instruction can be considered to be de-facto committed when it has reached the Point of No Return (PONR). From this point on, the instruction is guaranteed to commit. Therefore, the result of an instruction at or beyond the PONR can be treated as the committed result. As with architectural speculation contracts for general purpose architectures (described in Chapter 3), CSC obviates side-channel attack concerns by preventing memory accesses to illegal addresses from being issued. This approach prevents illegal data from entering the core before it might be exfiltrated by a side-channel attack.

CHERI capabilities separately authorise data access and instruction fetch, so one may distinguish between *data-CSC* and *instruction-CSC*.

Data-CSC requires aggressive enforcement of both memory bounds and capability provenance, i.e., the valid derivation of capabilities. As described in Section 3.5, checking bounds before data-memory access is reasonable and can be done in parallel to memory translation[1], which is needed to prevent the original Meltdown attack [64]. Capability provenance is also reasonable to enforce, as data values are generally forwarded results of a valid data flow from the committed register state. With capability provenance, a valid capability is always derived from another valid capability. Nevertheless, these constraints might easily be violated if data-CSC is not an explicit design goal; CHERI-Toooba was found to be vulnerable in an instructive way, as shown in Section 4.6.

Instruction-CSC enforcement is challenging, as instruction addresses are generally predicted with no dependency on committed register state. Nevertheless, instruction-CSC is highly desirable, as it uses the PCC bounds metadata provided by CHERI executables to constrain execution to the current compartment – minimising opportunities for Spectre-style transient-execution attack gadgets.

Instruction-CSC and data-CSC are closely related, and a violation of one can lead to a subsequent violation of the other. For example, code capabilities also allow loading data by default. A misspeculated PC capability can easily be moved into a register, e.g., using the `AUIPCC` instruction[2] in CHERI-RISC-V [93] and can be used to load memory. Therefore, enforcement of instruction-CSC is highly desirable in high-performance CHERI implementations. The initial CHERI-Toooba implementation did not enforce instruction-CSC, as I demonstrate in Section 4.5.2 However, Arm's Morello partially enforced instruction-CSC with PCC bounds forwarding at a high cost to performance. I propose to modify CHERI-Toooba to fully enforce instruction-CSC at a lower overhead, which is shown in Section 4.7.2. This mitigation mechanism called SinglePCC removes all speculative PCC state from the pipeline and only uses architecturally committed state to check PC bounds.

## 4.2 Definitions

In this section, I define the vocabulary used throughout this chapter. The definitions are built upon the vocabulary already defined in Section 3.2.

The contract for capability speculation argues about the *committed register state* in a processor. While determining the committed register state in an in-order processor is easy,

[1] Checking whether a virtual address is mapped in the page table can be considered a bounds check.

[2] The `AUIPCC` instruction loads the current content of the PCC register, adds an immediate to the address and loads the result into a general-purpose capability register. For more information, see the CHERI-RISC-V ISA reference manual [93].

this is not as straightforward in an out-of-order processor. I define a value part of the *committed register state* if the instruction producing this value is at or beyond the PONR, which means that this instruction cannot be cancelled anymore[3]. An instruction at or beyond the PONR is guaranteed to commit, but might not be at the Commit pipeline stage yet. This definition allows me to construct a capability contract with security guarantees, but equally permits for as much microarchitectural design freedom as possible in order to optimise for performance, e.g., forwarding of values produced by instructions at or beyond the PONR.

An *issued* memory access allows an address to produce any data result. For example, in an out-of-order processor, when a cached memory address is delivered to the load queue, the operation is allowed to proceed to index various structures, including the store queue, the store buffer, and the L1 data cache. If any indexing occurs based on the address, I consider the memory load *issued*.

## 4.3 Security evaluation

I must evaluate my contracts' ability to enforce CHERI's security guarantees in the face of transient execution. Published CHERI guarantees comprise: *bounds*, *permissions*, *encapsulation*, *provenance validity*, *monotonicity*, and *integrity* [93]. These invariants need to be upheld at all times, including in speculative execution. Failure to uphold these invariants can lead to vulnerabilities and attacks as demonstrated in Sections 4.5 and 4.6.

CSC described in Section 4.1 does not argue directly about speculative capability rights, but about speculative memory accesses. CSC states that memory accesses must be authorised by architectural capabilities (i.e., capabilities in the committed register state), but does not otherwise limit the existence of illegal capabilities in speculation. While this freedom is likely to be appreciated by microarchitects who are optimising for performance, I must demonstrate that the stated property is sufficient to maintain CHERI invariants in speculation. Consider the following example:

```
cincoffset ca0, ca1, a0 // Increase the offset out of bounds
clb a0, 0(ca0)          // Load from the previously
                        // manipulated capability
```

This CHERI-RISC-V assembly code loads a byte from an array at an offset ( `arr[i]` in C). Following my contract, this instruction can speculatively produce a valid capability even if the offset is out-of-bounds or is unrepresentable [103]. Equally, the `CIncOffset` instruction can speculatively pick up a capability from some microarchitectural state, e.g., sophisticated value prediction [20]. Thus, following the capability speculation contract, the `CIncOffset` instruction can speculatively execute before any of the instructions preceding it in program

[3] An instruction at or beyond the PONR is often also called *non-squashable*

order. This speculation, however, must be resolved before `clb a0, 0(ca0)` executes in order to avoid violating CSC by issuing an illegal memory access. This level of constraint must be sufficient to enforce all expected CHERI security properties.

CHERI *bounds* and *permissions* primarily relate to memory, and direct violations of these cases (e.g., an out-of-bounds load) are trivially excluded by CSC. It could be possible to transiently manipulate permissions to allow unsealing without causing an illegal memory access. However, the only advantage of an unsealed capability is to access memory, which is prevented by CSC. Therefore, even an illegally unsealed capability cannot expose new data to the core unless it can by used to access memory. Furthermore, a sealed capability is not secret. Thus, its fields can be inspected in speculation.

*Provenance validity* ensures that valid capabilities are always derived from other valid capabilities; *integrity* ensures that all such derivations are valid, in particular enforcing *monotonicity* such that derived capabilities never have greater privilege than their forbears. Similarly to above, if no memory access is allowed through fabricated or expanded capabilities, then no advantage can be gained from transiently possessing provenance-breaking or non-monotonic capabilities.

*Encapsulation* is the mechanism CHERI provides for software compartmentalisation, and relies on sealing permissions mentioned above. Changing compartments enables both executing instructions and accessing data from the new compartment, both of which require memory access. Thus, an attacker cannot gain any advantage by unsealing code and data transiently without violating CSC. Speculation on domain crossings is further discussed in Chapter 7.

Thus, the memory-centric specification of CSC is sufficient to uphold CHERI security guarantees while allowing transient register-to-register violations. Incidentally, however, the path I propose to take for microarchitectural enforcement of data-CSC was to guard all capability transformations as discussed in Section 4.7. This is not likely to be true for all microarchitectures, as extra performance can be gained from optimistically forwarding a result before all checks at the cost of the complexity of validating checks before issuing the memory request.

## 4.4 Evaluating CSC for non-capability architectures

In this section, I evaluate the security guarantees of my contract against transient-execution attacks as they are known on conventional systems. As before, I follow the classification of transient-execution attacks as presented by previous work of Canella et al. [22].

| | Inter-Comp |
|---|---|
| Spectre-PHT | ✓ |
| Spectre-BTB | ✓ |
| Spectre-RSB | ✓ |
| Spectre-STL | ✗ |

Table 4.1: Effectiveness of CSC against Spectre-like attacks. A (✓) means that the threat of the corresponding attack class is significantly limited by CSC.

### 4.4.1 Spectre attacks

I consider four different Spectre-style attacks for this evaluation: Spectre-PHT, Spectre-BTB, Spectre-RSB, and Spectre-STL. In the following paragraphs, I evaluate which security guarantees the capability speculation contract gives for each Spectre-style attack.

Spectre-PHT mistrains the branch direction predictor such that a security check, e.g., an array out-of-bounds check, is speculatively skipped. With CHERI invariants being enforced, an out-of-bounds access is forbidden and mitigated. The branch direction predictor is allowed to speculate, but the speculation will be stopped when the out-of-bounds capability is used for a memory access. Please note that this does not mitigate all attacks that rely on Spectre-PHT, e.g., wrongly predicting out of a loop is still possible [86].

Spectre-BTB and Spectre-RSB are attacks that are similar in the way they function. They both exploit mistraining microarchitectural structures that are responsible for branch target prediction – the BTB and the RSB. With CSC being enforced, speculation is only allowed to transitively reachable capabilities. This mitigates the most egregious forms of Spectre-BTB and Spectre-RSB attacks: attacks speculating to another protection domain and leaking secret data. Please note that this does not prevent speculation to the wrong code capability as long as it is present in the architectural register file. Furthermore, CSC does not have precise timing guarantees and thus exploiting early protection domain switches as demonstrated by Shivakumar et al. [86] is still feasible.

Spectre-STL enables attackers to read stale data by confusing memory disambiguation prediction to wrongly schedule a load before an architecturally older store to the same memory address. This attack is not mitigated by CSC because the attacker at all times has an architecturally valid capability to memory. The attack uses the change of order in performing the memory operations.

The results of this evaluation are summarised in Table 4.1. My contract is able to limit the attack surface for all attack classes, but Spectre-STL attacks. Memory versioning is not only independent of CHERI, but also inherent to all sophisticated microarchitectures. Therefore, it should be addressed with more general contracts as suggested in Section 3.8.1.

### 4.4.2 Meltdown attacks

CSC is able to fully mitigate Meltdown-US – the original Meltdown attack [64]. Due to the transitive closure of capabilities, my contract prevents any speculative access to other pages including kernel pages. Thus, the typical Meltdown attack scenario of dumping kernel memory from user space is successfully mitigated by my contract.

In general, Meltdown-style attacks exploit late permission checking in out-of-order pipelines. In contrast to Spectre-style attacks, Meltdown-style attacks are often architecture-specific and target a specific feature of the processor's ISA, e.g., Meltdown-NM [88] exploits the lazy usage of the floating point unit that is allowed in the x86 ISA. My capability speculation contracts are ISA-independent and cannot argue about Meltdown-style attacks that target certain ISA features. Therefore, instruction-set architects will need to design contracts tailored to their architectures in order to prevent such specific attacks.

### 4.4.3 Microarchitectural data sampling attacks

Microarchitectural data sampling attacks, e.g., Fallout [21] or ZombieLoad [85], are based on speculative execution that accesses arbitrary data from various sources inside as well as outside the core. CSC forbids any data to enter the core other than the data that can be reached from the architectural register state as well as all capabilities reachable through it. A processor adhering to the capability speculation contract will not be vulnerable to any microarchitectural data sampling attack because these data are not in the transitive closure of architectural capabilities. Please note that CSC does not argue about the temporal relation of memory accesses as long as they are allowed by the architectural register file state. Any attacks based on this are still allowed by CSC.

## 4.5 Hardware testing

For any speculation contract to be useful, it must be verifiable during hardware development. I evaluate the testability of using generators in the TestRIG framework, which injects instruction sequences into CHERI-Toooba and collects traces of the resulting execution, allowing assertions on its behaviour [50].

These test generators uncovered several violations of CSC in the initial CHERI-Toooba implementation. The data-CSC violations in particular allowed bypassing CHERI protection using transient execution to read any memory in the address space. These vulnerabilities were first identified in previous work conducting research on targeting CHERI-Toooba with transient-execution attacks [34]. This thesis contributes to automatically detect these attacks as well as to construct a full-system exploit in Section 4.6.

### 4.5.1 Data-CSC testing

To test data-CSC, I developed a single TestRIG generator that was able to produce examples of all known data-CSC violations in the CHERI-Toooba core. This generator arranges for the data-cache miss counter to indicate accesses not authorised by capabilities in the committed register file. The idea arose during a chat between Dr Jonathan Woodruff and me. Dr Woodruff started with an initial implementation of the generator, which was incomplete and did not produce all known violations. I re-wrote large chunks of this generator and completed its implementation. In the generator, each sequence starts with a full reset, which clears all caches. A prelude then prepares a capability granting access to a single word of memory, and loads that word. Henceforth, any cache misses will indicate a memory access not allowed by this capability. A random stream of capability instructions is then fed to the processor, followed by a read of the data-cache miss counter. If the counter shows unexpected misses, a violation is reported.

The three initialised operand registers used in the examples below are:

- **rOneWord**: the original, 1-word capability.
- **rInval**: a capability pointing to another location, but with the tag cleared.
- **rDest**: the destination register of all instructions, and the address of any loads.

A register operand used as a capability will be prepended with 'c' and a register operand used as an integer will be prepended with 'r', e.g., **cDest** stands for a capability destination register. Following are two counterexamples:

#### CBuildCap

```
clb rDest, cDest[0]                 // Load delaying the
                                    // following instructions
                                    // from committing
cbuildcap cDest, rInval, cInval     // Transiently building a
                                    // capability
clb rDest, cDest[0]                 // Loading a secret
```

While the capability load is waiting to commit, `CBuildCap` transiently constructs a valid capability from an untagged value and loads through that forged capability. This load will miss the L1 data cache and thus increase the miss counter indicating a violation.

### CSetBounds

```
csetbounds cDest, cOneWord, rDest       // Transiently extend
                                        // bounds
cincoffsetimmediate cDest, cDest, 2903  // Move pointer to
                                        // disallowed address
clb rDest, cDest[0]                     // Loading a secret
clb rDest, cDest[0]                     // Delay for
                                        // measurement
```

Transiently set the length past the current bound, move the pointer into the illegal frontier, and perform a load, with an additional load to delay for measurement. This delay is necessary because the counter increase caused by the first load is not caught if the counter reading instruction immediately follows[4]. As with the counterexample above, this load will miss the L1 data cache and thus increase the miss counter. Similar counterexamples are produced for `CSetBoundsImmediate` and `CSetBoundsExact`.

In addition, this generator produces counterexamples for `CUnseal` and `CInvoke` that illegally dereference sealed capabilities in transient execution. Both counterexamples can be used to break compartmentalisation in a CHERI system and therefore constitute a major violation of the CHERI security model. The compartmentalisation approaches in CHERI are more closely described in Chapter 5.

### Analysis of data-CSC violations

These data-CSC violations allow trivial circumvention of capability protection using transient execution. I constructed examples that set the bounds of a capability to the maximum integer value, and, while waiting for the exception to commit, loaded a byte through this forged capability and used it as the offset of a second transient memory access, transmitting the value to a cache side channel. Forwarding undefined values in the pipeline in a faulting condition is a classic Meltdown-style vulnerability, similar to forwarding data during a page permissions fault in the original Meltdown attack [64]. Previous work [34] dubbed this family of transient-execution vulnerabilities collectively as Meltdown-CF: Meltdown Capability Forgery.

As elaborated in previous work [34], the data-CSC violations in CHERI-Toooba are due to forwarding results in the pipeline that do not account for exception checks. Several crucial CHERI instructions require a full bounds check, which is not performed until the cycle after the execution result has been forwarded. As shown in Figure 4.1, the trap code is set in the Re-Order Buffer (ROB) and the exception is only taken once the trapping instruction reaches the head of the ROB and would commit.

[4]The L1 data cache miss counter is a microarchitectural state counter whose implementation is specific to every microarchitecture.

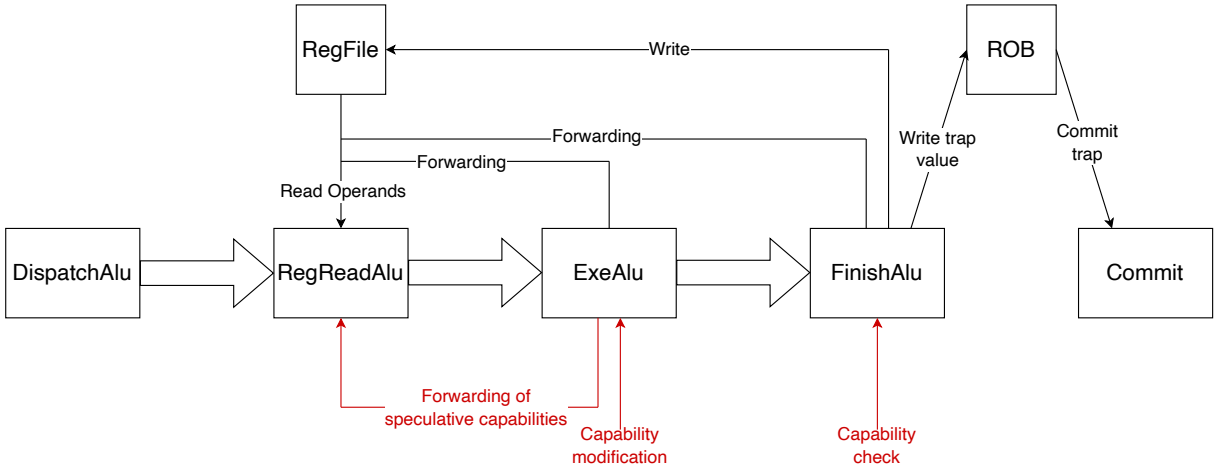



Figure 4.1: The CHERI-Toooba ALU pipeline with late exceptions checks.

I evaluate the seriousness Meltdown-CF with a user-level exploit in a pure-capability process on CheriBSD in Section 4.6, and present as well as evaluate a solution in Section 4.7.1.

### 4.5.2 Instruction-CSC testing

Before looking at instruction-CSC testing, it is important to understand how branch prediction with capabilities works on CHERI-Toooba. When predicting indirect jumps, one has the choice to predict either the PC itself or the entire PCC including its bounds. The latter leads to good performance because the entire code capability is present and can be used for subsequent instructions, e.g., the `AUIPCC` instruction needs to read the code capability's bounds. However, this approach also bears dangers as shown in this chapter. Furthermore, full bounds prediction brings an increased area cost as BTB and RSB entries need to be able to hold entire capabilities instead of addresses. Direct branches and direct jumps only change the offset of the PCC and thus do not need to predict the bounds. CHERI-Toooba as tested in this chapter employs full bounds prediction.

Instruction-CSC is violated if an implementation executes instructions not allowed by the committed PCC and register state. My instruction-CSC counterexample generator trains the BTB with a sequence of jumps within a PCC with permissive bounds. It then restricts the bounds of the PCC and executes a similar sequence to trigger mispredictions with permissive bounds.

Instruction-CSC allows instructions to be *fetched* through wild PCCs. Thus, one must detect if they are *executed*. Only an instruction that executes takes microarchitectural effect and thus can constitute a transient-execution vulnerability. In this generator, I used the `AUIPCC` instruction, which copies the current PCC into a register, as a portable mechanism for observing the PCC in Execute without requiring special-purpose coun-

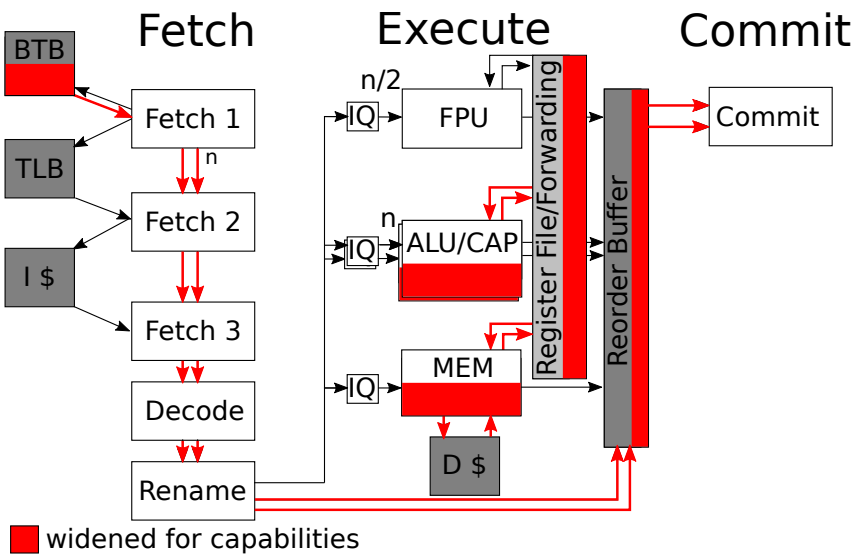



Figure 4.2: Pipeline diagram of CHERI-Toooba with n=2.

ters. The following is a reduced counterexample from my instruction-CSC generator:

```
cjalr x0, c23    // Jump to speculative capability
auipcc c25, 0    // Load current PCC to register
clb x24, c25[0] // Load a secret
```

This counterexample triggers a misprediction with `CJALR`, and then with `AUIPCC` loads the current PCC into a register. This capability is then used to perform a load (`clb`) – allowing me to count unexpected data cache misses, as with the data-CSC generator. This strategy actually violates data-CSC using an instruction-CSC violation.

## Analysis of instruction-CSC violations

CHERI-Toooba violates instruction-CSC due to predicting the entirety of the PCC, including the bounds and permissions, as depicted in Figure 4.2 by the red extension of the BTB with red paths through the Fetch pipeline stages [82, 83]. While this design minimises performance overhead relative to the base Toooba microarchitecture, it allows transient execution into foreign compartments.

Jump target prediction is critical for performance of superscalar and out-of-order pipelines, so instruction-CSC enforcement is particularly performance-sensitive. Because these pipelines are superscalar, they can lose a large number of instructions on a flush, and because they are out-of-order, they can leverage PC prediction to reorder instructions from different basic blocks. Two microarchitectural solutions for solving instruction-CSC violations are presented with implementations and performance overheads in Section 4.7.2.

| Instruction | Violation |
|---|---|
| CBuildCap | data-CSC, crafting a forbidden capability |
| CSetBounds | data-CSC, extending capability bounds |
| CInvoke | data-CSC, jump to forbidden capability pair |
| CUnseal | data-CSC, unsealing a capability without permissions |
| CJALR | instruction-CSC, jumping to a foreign capability |

Table 4.2: Overview of testing CHERI-Toooba against CSC.

| *CBuildCap* | *CSetBounds* | *CSetBounds Exact* | *CSetBounds Immediate* | *CUnseal* | *CInvoke* |
|---|---|---|---|---|---|
| 0.10% | 0.05% | 0.20% | 0.20% | 0.20% | 1.80% |

Table 4.3: Distribution of sequences that produce counterexamples for the data-CSC generator during 2000 runs. Overall, 2.55% of runs produced a counterexample.

### 4.5.3 Testing overview

My testing found both violations against data-CSC and instruction-CSC in CHERI-Toooba. The results are summarised in Table 4.2. These violations can lead to dangerous attacks as demonstrated in Section 4.6.

### 4.5.4 Evaluation of instruction generators

My test generators are efficient at discovering CSC counterexamples by using smart shrinking after detecting an initial counterexample. As described in Section 4.5.1, the data-CSC generator produces the six classes of counterexamples listed in Table 4.3, with an overall probability of 2.55% of discovering a counterexample for any sequence in 2000 runs. The skew towards the `CInvoke` counterexamples can be explained by a bias in the shrinking mechanism of TestRIG [50, 79]. The instruction-CSC generator produced one class of counterexamples with a chance of 0.35% measured in 2000 runs.

## 4.6 Meltdown-CF attack

In this section, I present a successful Meltdown-CF attack on the CHERI-RISC-V platform I used for the previous evaluation. I show how attackers can make use of the Meltdown-CF vulnerability, and finally I present an evaluation of my attack. The attack is inspired by the results I obtained in previous research [34].

### 4.6.1 Attack setup

I run all experiments on the CHERI-Toooba platform presented in Section 2.11. My goal in this attack is to show how Meltdown-CF can in speculation break one of the fundamental security guarantees of CHERI – spatial memory safety [93]. My setup is that two protection units are executing in the same address space, which is a scenario commonly used when implementing compartmentalisation on CHERI processors, e.g., as used by library compartmentalisation [38]. More information on lightweight compartmentalisation on CHERI can be found in Chapter 5. In my attack, one protection unit attempts to access the memory of another protection unit without having a valid capability to it.

### 4.6.2 Conducting the attack

For this sample attack, I decided to use the Meltdown-CF vulnerability connected to the `CBuildCap` instruction as presented in Section 4.5.1. However, one could have used any other variant presented in Section 4.5 to demonstrate that one can break CHERI's spatial memory safety in speculation. My approach is to build a capability from scratch in speculation, while the code I am running is architecturally not allowed to build that capability. This illegal capability permits access to memory of the other protection unit in my attack setup. Then I use this capability to access a secret and encode it in the cache side channel.

First, I need to assemble the bit pattern of the capability, which I can achieve by writing two 64 bit words to memory. This will result in an invalid – because untagged – capability. The goal now is to achieve that this capability is tagged in speculation, which can be facilitated because the `CBuildCap` instruction in CHERI-Toooba is implemented such that it first builds the capability and later checks for a possible exception (see Figure 4.1). I then use this forwarded capability and access the memory it points to. Next, I encode the data through the cache by accessing an array and probing it later. This gives me the fundamental building block of my attack, but I need further building blocks to make it work on a real-world operating system.

The protection unit in the process I want to attack is running for a longer period of time and one needs to keep the process alive. In many attack scenarios the secret might

```
meltdown_cf:
  clb t0, 0(ca1)           // Load variable to branch on
  blt x0, t0, end          // Branch mispredicted to next
                           // instruction
  cbuildcap ct1, ca1, ca0  // Generate speculative capability
  clb t2, 0(ct1)           // Load secret
  and t2, t2, a6           // Apply bit mask to get subset of
                           // secret
  sll t2, t2, a7           // Shift secret to achieve cache
                           // line granularity
  cincoffset ct2, ca5, t2  // Add offset to probing buffer
  clb t2, 0(ct2)           // Transmit secret via load to
                           // buffer
end:
  cret                     // Return to caller
```

Listing 4.1: Meltdown-CF attack code.

change from one process instantiation to another process instantiation. However, executing the `CBuildCap` instruction as is would lead to an exception and terminate the process. I take inspiration from previous Meltdown-style attacks and embed the code in an `if` statement [22]. This way, the exception caused by `CBuildCap` is only entered through control-flow misspeculation and thus will not lead to the process being terminated by the operating system. Guarding the attack by an `if` statement adds the requirement that I need to train the branch direction predictor behind the `if` statement. Therefore, my attack contains training blocks before I can start an exploit call to the code containing the `CBuildCap` instruction. The core part of the attack code is depicted in Listing 4.1.

The other big building block is flushing the caches, which is required due to two reasons. First, I need to generate a high signal-to-noise ratio for the cache side-channel attack. In order to achieve that I need to evict all cache lines of the buffer that I use to communicate which data I accessed speculatively during an attack run. My flushing and probing mechanisms required particular care in the attack development due to lazy allocation of pages by the operating systems as well as optimisations used by the compiler. Second, I need to ensure that the misprediction of the `if` statement guarding the `CBuildCap` instruction is resolved as late as possible. Thus, I make the `if` statement dependent on a load that will miss all caches.

In summary, a Meltdown-CF attack requires the fine-grained interplay of multiple different steps. An overview of the different attack steps is presented in Figure 4.3.

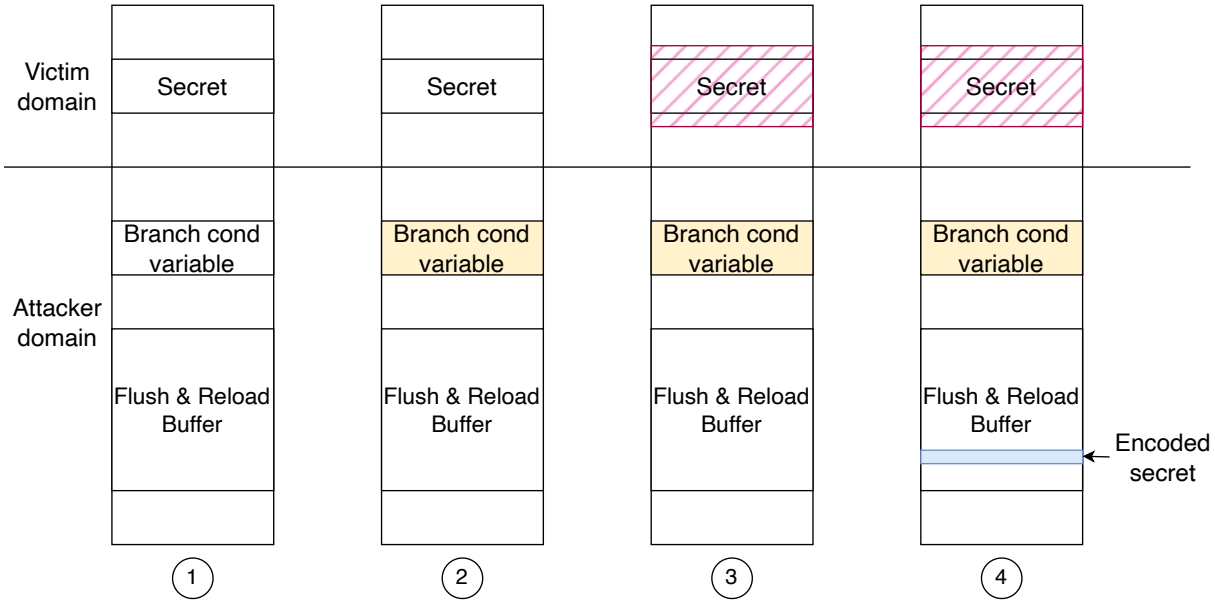



Figure 4.3: This figure summarises the steps for the Meltdown-CF attack. The first diagram shows the system in the start state, where the attacker and victim are in two different protection domains. In the second diagram, the access to the branch cond variable will miss all caches to delay resolution of misspeculation. In the third diagram, the attacker creates a speculative capability covering the memory where the secret is stored. In the final diagram, the attacker accesses the secret and transmits it by loading a cache line.

### 4.6.3 Evaluation

I ran the attack on the DE10 platform, which can run a 50MHz clocked bitfile of CHERI-Toooba, which is running the CheriBSD operating system [28]. I measured a reading rate of approximately 1.4 bytes/second for my attack, with an error rate of 0%, which means that I read out the correct secret in every attack run. On an industry-scale processor, I expect the attack to significantly scale up in speed. The original Meltdown attack was conducted on multiple processors with cores being able to reach frequencies up to 4.7 GHz [64]. My attack demonstrates that Meltdown-CF vulnerabilities can be applied to running CHERI systems in the field. My attack or similar ones can be used to considerably harm CHERI applications running on vulnerable processors.

## 4.7 CSC enforcement

The TestRIG generators found that CHERI-Toooba did not enforce CSC, but had multiple violations of both data-CSC and instruction-CSC. I defined CSC with the expectation that a superscalar, out-of-order processor can be compliant without sacrificing performance; thus it is necessary to demonstrate that enforcing CSC in CHERI-Toooba is possible at full performance.

The Arm Morello [41] implementation of CHERI provides an instructive comparison with CHERI-Toooba. Morello is a 4-way superscalar, out-of-order CHERI implementation based on the Arm architecture. Morello made several decisions in the architecture and microarchitecture which resulted in being less vulnerable to CSC violations, though in some cases this came at a prohibitive performance cost.

### 4.7.1 Data-CSC enforcement

As described in Section 4.5.1, CHERI-Toooba is vulnerable to Meltdown-CF due to forwarding capabilities before exception checks. The CHERI-Toooba exception check is split into two pipeline stages: ExeAlu and FinishAlu as depicted in Figure 4.1. Result values are forwarded from ExeAlu, accelerating the common case, but the exception is reported in FinishAlu, marking the instruction for cancellation at Commit. While one could choose to detect the illegal operation early and forward an invalid capability in this case, the faulting semantics for these instructions actually imply dangerous microarchitectural design. It is a common microarchitectural pattern to register exceptions in the ROB and let execution go on until the faulting instruction is retired. This pattern also led to processors being vulnerable to the original Meltdown attack [64]. The architectural notion of having faulting instructions instead of instructions that forward untagged capabilities suggest this dangerous microarchitectural design pattern as it was implemented on CHERI-Toooba.

Arm's Morello architecture took another approach which encourages data-CSC-safe implementations. The Arm architecture avoids exceptions on arithmetic operations, and in keeping with this principle, the Morello architecture simply clears the capability tag from the result of illegal capability modification instructions [6, 63]. This implies that implementations should reflect failure atomically in the forwarded result, removing the opportunity for an illegal capability to be introduced in transient execution.

Like CHERI-Toooba, Arm's Morello implementation also requires two cycles to perform a full bounds check along with capability modification [41]. Morello therefore performs `CSetBounds` and `CBuildCap` in the multi-cycle arithmetic pipeline along with multiply and divide, increasing the latency of these instructions. These instructions are rarer than `CIncOffset`, which is used for pointer arithmetic and requires a simpler representability check [103], so this is not a major overhead.

While I could propose leaving the architecture unchanged and microarchitecturally poisoning illegal capability values, I propose to change the CHERI-RISC-V specification to clear tags on capability violations to suggest microarchitectures that conform to data-CSC, preventing Meltdown-CF vulnerabilities. Dr Peter Rugg contributed this change to the CHERI-RISC-V specification [93], and implemented the change in CHERI-Toooba. To move CHERI-Toooba to tag-clearing semantics, the entire exception check was moved to the ExeAlu stage so that the forwarded tag of the result could reflect any error conditions, eliminating all Meltdown-CF violations. This modified design did not change resource

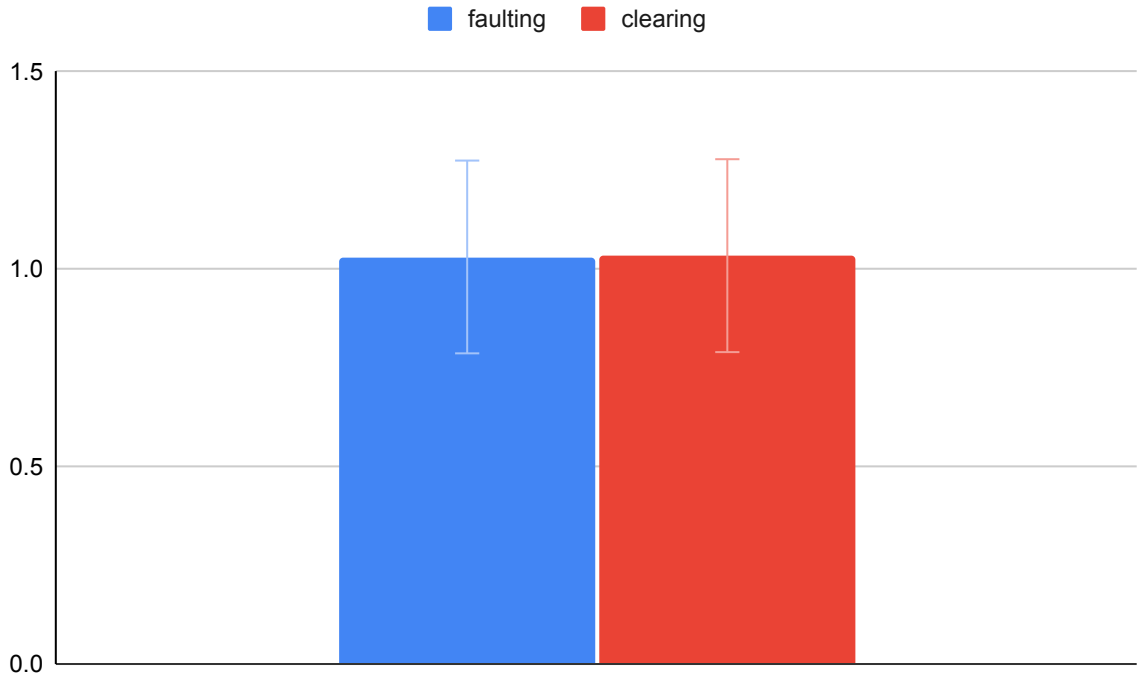



Figure 4.4: The cycles-per-instruction distribution across the SPECint2006 benchmarks is unchanged when enforcing data-CSC using tag clearing, demonstrating that performance is unaffected. The average denotes the arithmetic mean of all SPECint2006 benchmarks.

utilisation on the VCU118 implementation, with both Lookup Table (LUT) usage (mean +0.86%) and FF usage (mean -0.5%) for the cores within the range of synthesis variation. When running SPECint2006 on CHERI-Toooba enforcing with tag clearing compared to the faulting version, the results are very similar to each other. The measurements show a CPI improvement by approximately 0.3% for tag clearing. The results are depicted in Figure 4.4. While this design passed timing at 25MHz on the VCU118 prototype and at 50MHz on the DE10 prototype, a high-performance ASIC implementation may need to implement rare capability operations using multiple cycles.

A version of CHERI-RISC-V with my proposed tag-clearing semantics is now in the process of being ratified by RISC-V International[5], and is being implemented in commercial processors [25].

[5]See GitHub repository: `https://github.com/riscv/riscv-cheri`.

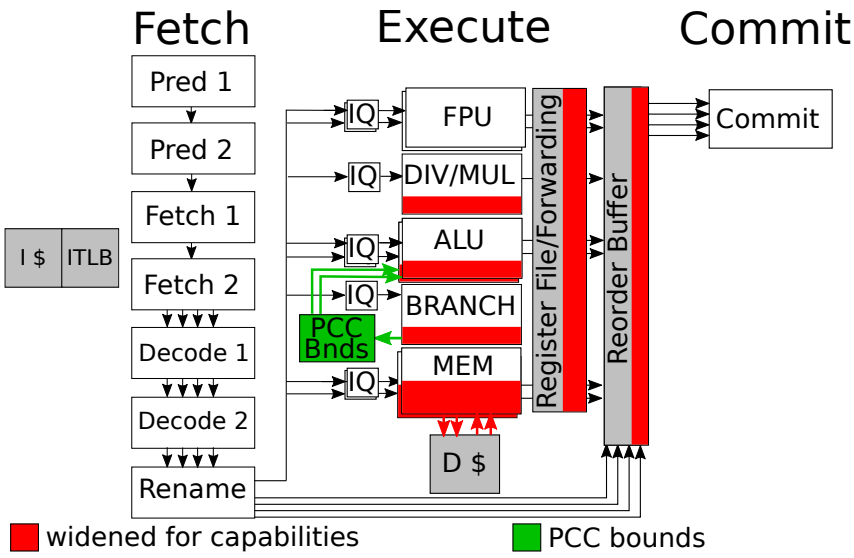



Figure 4.5: Approximation of the Morello microarchitecture, featuring PCC bounds forwarding. This diagram was drawn by Dr Jonathan Woodruff and is based on public information available on the Neoverse N1 [62] CPU pipeline.

### 4.7.2 Instruction-CSC enforcement

CHERI-Toooba stores the bounds of previous PCC targets in the BTB and freely predicts PCC bounds when the Fetch stage encounters a jump. This provides ideal performance, but leads to violations of instruction-CSC, as foreign targets with their own bounds are reintroduced from the BTB, allowing speculative execution to wander between domains. The original CHERI-MIPS implementation [105] provides a solution to this problem in an in-order core by predicting only the address of the PCC, but *forwarding* the bounds from the last-executed branch. This is possible because the bounds of the PCC are not needed in the early stages of the pipeline, but affect only exception conditions or register writes (e.g., when linking). Thus, an in-order core can forward PCC bounds for efficiency and performance, while incidentally enforcing instruction-CSC.

Out-of-order and superscalar implementations require far more complexity to forward bounds while maintaining consistency between the two halves of the PCC. Nevertheless, the Morello CHERI microarchitecture takes such an approach. To avoid doubling the size of prediction structures, Morello predicts only the address, as with CHERI-MIPS, and maintains a table to hold unique bounds [94]. Unfortunately, PCC readers must block until the last jump is executed. This approach still allows wild execution, but prevents wild PCCs from being used for data access; that is, instruction-CSC cannot lead to a violation of data-CSC. However, the Morello design still allows for instruction-CSC violations[6]. Figure 4.5 illustrates an approximation to the Morello design, which features bounds forwarding.

[6]This is because in Morello, instructions can still be speculatively fetched.

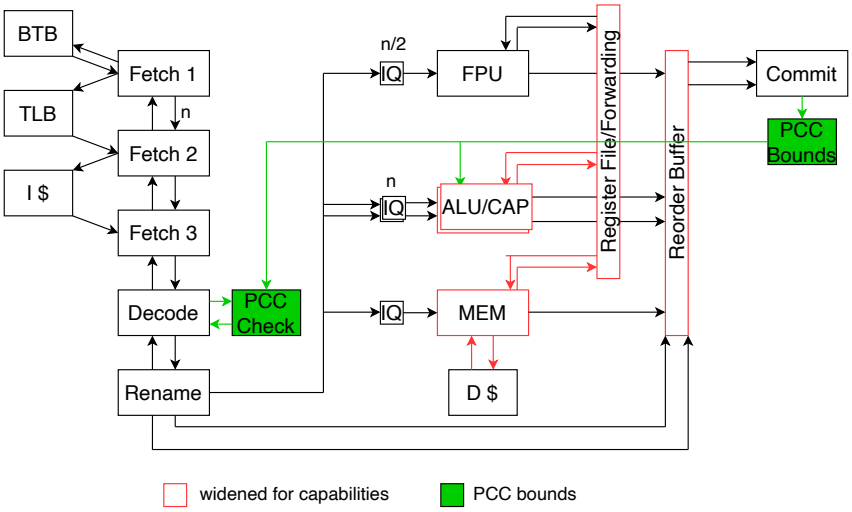



Figure 4.6: CHERI-Toooba with SinglePCC prediction, n=2.

A straightforward, safer, and higher-performance strategy for enforcing instruction-CSC is to simply predict that the bounds of the PCC did not change. If one allows only those instructions to execute that lie within the PCC written by the latest-executed branch (which in turn must be derived legally according to data-CSC), then an implementation will comply with instruction-CSC.

I call this implementation strategy *SinglePCC*, illustrated in Figure 4.6. I discussed this idea with Dr Jonathan Woodruff who carried out an initial implementation. For SinglePCC, we removed bounds from all program counter state everywhere in the pipeline and replaced them with a single PCC register. Any logic in the pipeline that needs the complete PCC simply appends the bounds from the PCC register to the current instruction address, thus speculating that the bounds of the PCC have not changed. Any jump to a capability with different bounds will trigger a flush at the Commit stage to ensure all older instructions commit with the correct bounds.

I synthesised the SinglePCC design for two FPGA boards: the Xilinx VCU118 board and the Terasic DE10Pro board. I evaluated our implementation in two ways: resource usage on the FPGA board and performance running well-established benchmarks for microprocessors. SinglePCC simplifies the design by deriving all instances of PCC bounds in the pipeline from a single register, reducing register and LUT elements used. I measure a reduction of approximately 4.89% for LUTs on both boards as well as 5.48% and 6.67% fewer registers on the DE10 board and VCU118 board respectively. The results are summarised in Figure 4.9. This constitutes a significant reduction in area, eliminating over 15% of the logic required to add CHERI support to the Toooba core [82, 83].

Figure 4.7 compares the cycle overhead of SinglePCC, which fully enforces instruction-CSC, against Morello's bounds forwarding, which prevents instruction-CSC violations from leading to data-CSC violations. Figure 4.7 shows the subset of the SPECint2006 benchmarks that did not need any source code modifications to run on CHERI. This choice was made for reproducability reasons and is described more closely by Watson

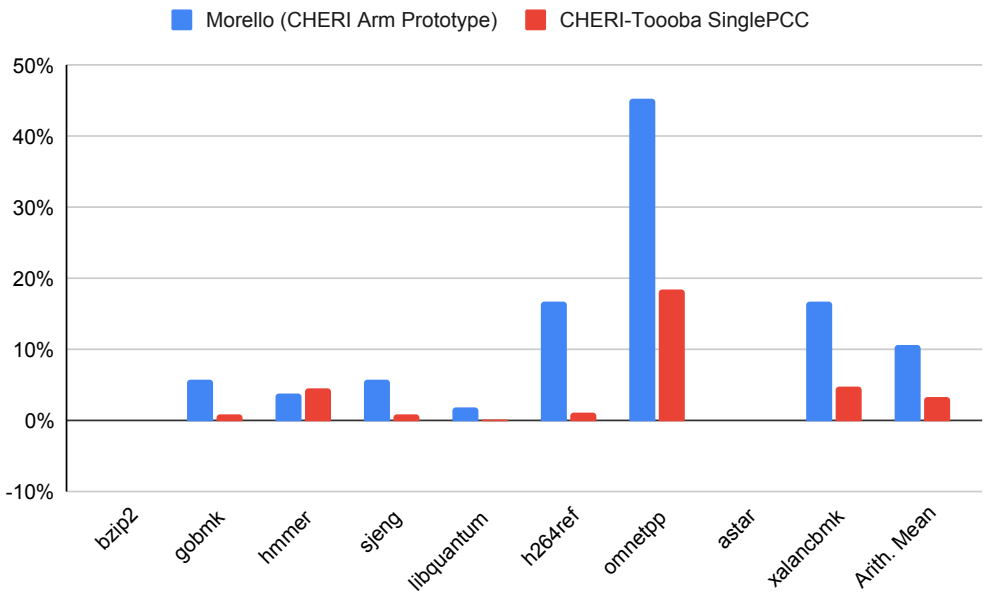



Figure 4.7: Cycle overhead of Morello bounds forwarding and the SinglePCC implementation on CHERI-Toooba. The diagram depicts the overhead of running the implementations against the respective CHERI purecap baseline. For Morello, this means running kernels with the purecap and benchmark ABI of which the latter does not have instructions that cause bounds forwarding.

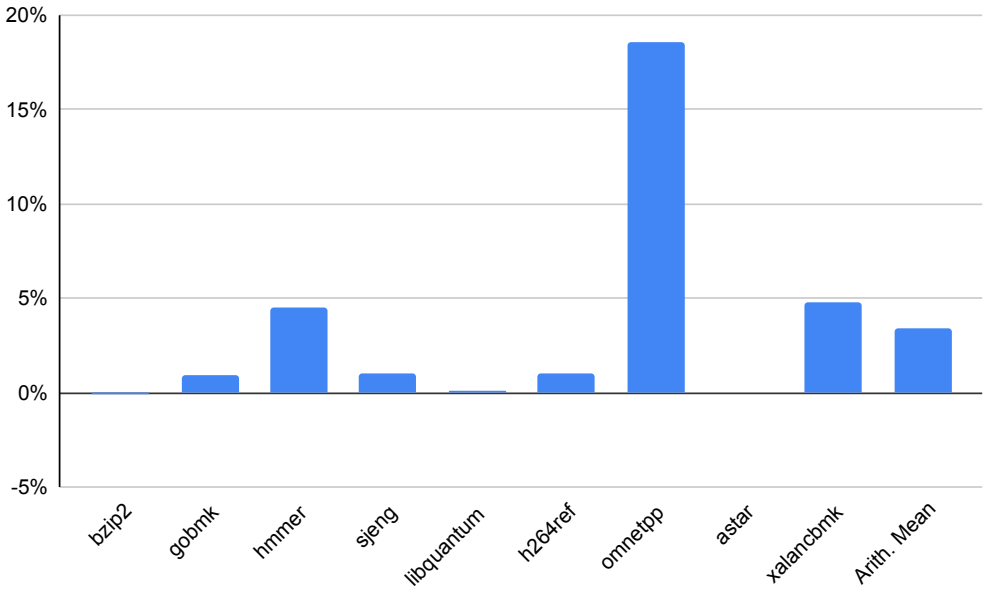



Figure 4.8: Extra pipeline flushes due to capability jumps that change bounds when using SinglePCC in SPECint2006 with distinct bounds for each dynamically loaded library.

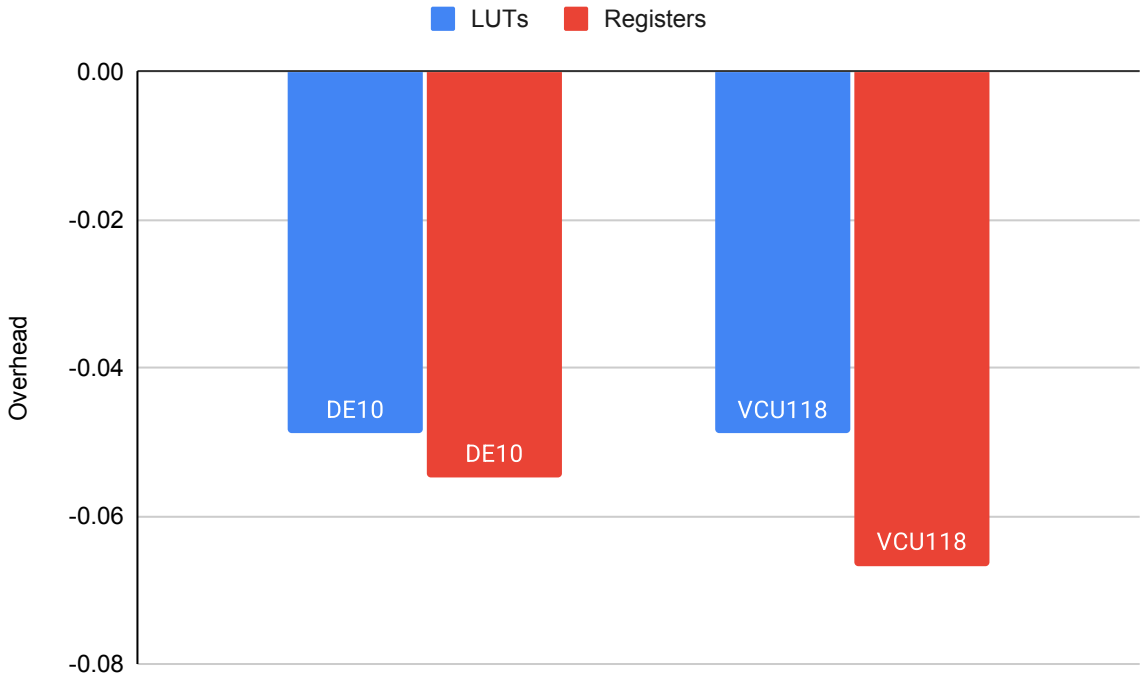



Figure 4.9: LUT and register overhead when synthesising the SinglePCC design for the DE10Pro and VCU118 FPGAs showing substantial improvements.

et al. [92]. The overhead for Morello's bounds forwarding is measured by a compiler mode called *Benchmark ABI* [92]. This compiler mode uses global bounds for all code capabilities, which apart from PCCs also includes all return capabilities. With this compiler mode, it is possible to measure the overhead of bounds forwarding. This sacrifices the control-flow-integrity guarantees of CHERI, but allows us to eliminate stalls on PCC bounds dependencies to measure performance without this limitation. Despite flushing on every cross-library call, SinglePCC incurs only 3.43% overhead on average, versus 10.7% for Morello's bounds forwarding. While a pipeline flush is much more expensive than a single pipeline dependency stall, reads of PCC values prove far more common in dynamic execution than cross-library jumps. Figure 4.8 shows the number of flushes per 1000 instructions in the benchmarks; only omnetpp sees more than 2 flushes per 1000 instructions, and is the only benchmark with a cycle overhead greater than 5%.

Future implementations have several options to achieve greater performance than SinglePCC while still respecting instruction-CSC. For example, rather than a full pipeline flush on every change of PCC bounds, one could predict jumps that would change the PCC, and pause the following instructions in Rename until the jump has committed. Assuming effective prediction, this should cut cycle overhead in half. Beyond this, more sophisticated solutions might develop a full bounds prediction engine that respects committed register state. It seems plausible that ASIC implementations can offer both performance and can implement CSC. The Morello processor was the first commercial implementation of CHERI and already implemented partial CSC enforcement while still offering good performance.

However, it is also important to note that Morello is only a prototype implementation with a limited budget. Further implementations are likely to see significant improvements.

## 4.8 Conclusions

In this chapter, I have articulated the Capability Speculation Contract (CSC) that precludes transient-execution attacks against CHERI protection, and have demonstrated complete enforcement at a 3.43% performance overhead in CHERI-Toooba, with hope for further optimisations in the future. The discovery of CSC violations in the CHERI-Toooba implementation, resulting in the Meltdown-CF vulnerability, is ample proof that such a clearly defined and testable contract is necessary to develop safe superscalar out-of-order CHERI processors. This work has paved the way for a standardised CHERI-RISC-V extension with tag-clearing semantics, to encourage implementations that are safe from Meltdown-CF.

My work gives evidence to Hypothesis H.3, which claims that the CHERI architecture can be extended using architectural speculation contracts that constrain the behaviour of the microarchitecture in speculation in order to avoid information leakage. My work has proved that an important part of microarchitectural leakage was stopped, but it does not prove that *all* leakage has been stopped, which remains future work. Furthermore, my work proves that Hypothesis H.4 is true. This hypothesis claims that a CHERI microarchitecture can be extended to implement the architectural speculation contracts and does not incur significant PPA overheads.

# Chapter 5

# Domain crossing mechanisms

In this chapter, I describe the domain crossing mechanisms that are laid out in the latest CHERI ISA reference manual. Furthermore, I describe the mechanisms specified in the Morello ISA with a special focus on the Executive-Restricted mechanism, which is used for lightweight compartmentalisation on Morello. The limitations of the Executive-Restricted Morello approach motivated me to explore a different approach to support library compartmentalisation on CHERI-RISC-V: Thread Identifiers (TIDs) as well as capability extension to them. In the final section of this chapter, I present research into enhancing cycle performance on common domain crossing patterns. My research on the TID approach presented in this chapter has led to a sub extension in the RISC-V CHERI specification that is currently on the path of ratification[1].

## 5.1 Background

### 5.1.1 Library compartmentalisation

Library compartmentalisation is a model developed by Dapeng Gao that puts each library into its own compartment [38]. When calling a function in a library from the 'main' program or from a different library, this transition will trigger a switch from the current compartment into the compartment of the called library function. The transition takes place through a *trampoline*, which takes care of switching out the caller compartment and sets up the called compartment. Once the called compartment has finished, it will return to the trampoline, which then does the switching the other way around. In library compartmentalisation, there exists a trampoline for every called target function. A trampoline provides control-flow guarantees: a `return` to the trampoline will always divert

[1]See GitHub repository: `https://github.com/riscv/riscv-cheri/releases`.

control-flow back to the caller. When consecutively applied, this leads to *well-bracketed* control flow[2].

The model by Dapeng Gao requires that the trampoline code is more privileged than the code running in the respective compartments. Running the trampoline code in a more privileged domain enables the trampoline code to access state that it needs to manage compartment transitions. This state must not be accessible to compartments and is protected by the underlying privilege mechanism.

Before this dissertation, library compartmentalisation only existed on the Morello platform. In the following text, I will evaluate existing mechanisms and motivate the approach I developed.

### 5.1.2 Compartmentalisation primitives in the CHERI ISA reference manual

The CHERI ISA reference manual [93] describes multiple primitives that facilitate domain crossings: sealed entries (sentries), sealed pairs of capabilities, and indirect sealed entries. The first two are implemented in CHERI-RISC-V whereas indirect sealed entries only appear in the *Experimental CHERI Extensions* chapter.

A sentry is a code capability, which can only be used to divert control flow to it. All other operations pose a CHERI violation and are handled via clearing the validity tag or throwing a CHERI exception.

A sealed capability pair consists of a code capability and a data capability, which is linked through a matching otype. Via a specific instruction, control flow is diverted to the code capability and the data capability is unsealed and put into a destination register. The number of concurrently existing sealed capability pairs is limited to the number of bits the otype is allocated in the capability format. In CHERI-RISC-V, the otype of 128-bit capabilities is currently set to 18 bits limiting the otype space to 262144 possibilities. Sentries and sealed pairs of capabilities are described in more depth in Section 2.9.

Indirect sealed entry capabilities are sealed capabilities that are linked together through memory instead of otypes. Thus, they are not subject to the finite otype space limitation, but rather limited through the memory being accessible to the system. This makes indirect sentries being scalable with the system size. Currently, two kinds of indirect sentries are described: points-to-PCC and points-to-pair. In the points-to-PCC variant, a sealed data capability points to a code capability in memory. In the case of points-to-pair, a sealed capability points to a pair of a code and data capability, which are stored adjacent in memory. The two variants of indirect sentries are visualised in Figure 5.1.

[2]Please note that a malicious compartment might still choose to not return at all, which does not break the model of well-bracketed control-flow though.

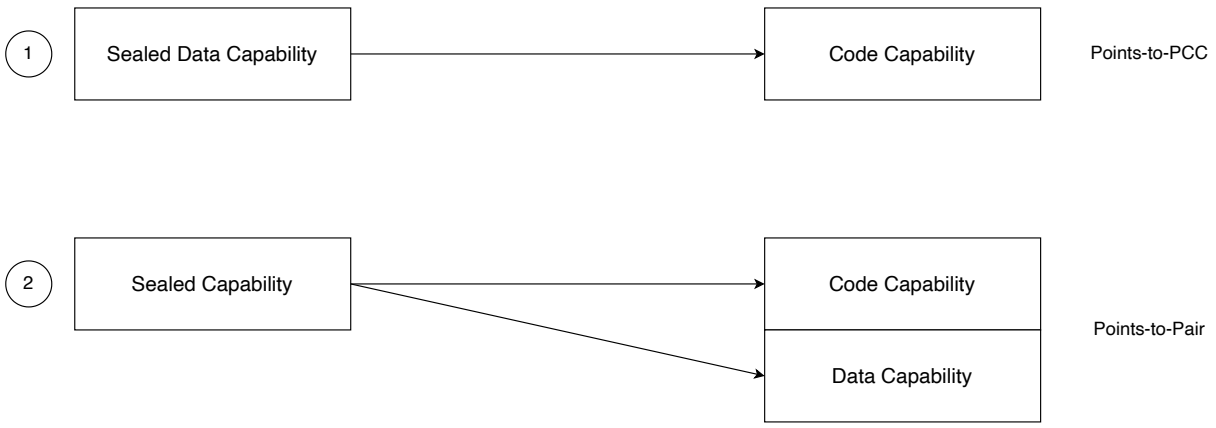



Figure 5.1: The CHERI ISA reference manual describes indirect sealed entries. The first diagram shows a *points-to-PCC* indirect sentry and the second diagram shows a *points-to-pair* indirect sentry.

### 5.1.3 Suitability for library-compartmentalisation

In the following text, I evaluate whether the compartmentalisation primitives presented in the CHERI ISA reference manual are suitable to support library compartmentalisation on CHERI-RISC-V.

#### 5.1.3.1 Sealed entries

Using solely a sealed entry for jumping to a trampoline is not sufficient to support library compartmentalisation. The sentry is a single code capability that does not come with a data capability that is needed for the privileged trampoline code. Therefore, sentries on their own cannot be used for library compartmentalisation. However as shown in Section 5.3, sentries can be an important building block to support library compartmentalisation on CHERI-RISC-V.

#### 5.1.3.2 Sealed pair of capabilities

A sealed pair of capabilities offers both a code and data capability. However, no matter which software thread calls the pair of capabilities, the thread will always receive the same data capability. To get around this problem, instead of calling the function directly, a function call first calls a trampoline, which eventually calls the target function. In the library compartmentalisation model, the same trampoline code is called by multiple threads[3] and operates in a way that is specific to the thread it was called by. In order to facilitate that the trampoline can work in a thread-specific way, the calling code needs to have a per-thread data capability[4]. For this to happen, the code would need to be

[3]Please note that all threads will have the same entry point into a trampoline.

[4]Please note that passing a capability to the thread itself will not work without a dedicated register because otherwise unprivileged code can alter the per-thread data capability, e.g., stash an old state of the capability pointer and hand it to the trampoline.

aware of which thread is executing, which is exactly the problem that is solved within this chapter. Therefore, a sealed pair of capabilities on its own is not sufficient to support library compartmentalisation on CHERI-RISC-V.

#### 5.1.3.3 Indirect sealed capabilities

Indirect sealed entries offer a code and data capability, while not being limited by the `otype` space. A sealed capability points to a region of memory, which then holds a code capability[5]. A dedicated instruction unseals the sealed capability into a dedicated register, loads the code capability, and installs the code capability into the PCC register. Like a conventional sealed pair of capabilities, it is not possible with indirect sealed entries to have a thread-dependent data capability for privileged code. This is a requirement of library-compartmentalisation However, indirect sealed entries offer interesting avenues for research on CHERI domain transitions as well as different compartmentalisation models. A compilation of early thoughts is presented in Appendix A.

Of all the domain transition primitives being described in the CHERI ISA reference model, none on its own is suitable to support the requirements of the current form of library compartmentalisation on CHERI-RISC-V. In the following section, I will present an additional domain transition mechanism being present in the Morello-ISA and evaluate its suitability for supporting library compartmentalisation on CHERI-RISC-V.

## 5.2 Morello compartmentalisation primitives

The Morello-ISA includes all the compartmentalisation transition primitives described in Section 5.1.2. In addition, the Morello-ISA also introduces a new primitive called the *Executive-Restricted* mode. In this section, I will describe this mechanism and evaluate whether it is suitable to support library compartmentalisation on CHERI-RISC-V.

### 5.2.1 Morello Executive-Restricted ISA

The Morello ISA introduced the *Executive* and *Restricted* mode [6, 41]. This splits the privilege levels into sub-privilege levels: the more privileged *Executive* mode and the less privileged *Restricted* mode. Arm's 64-bit A architecture defines four privilege levels, which are called *Exception Levels (ELs)* in Arm's terminology where EL0 is the least privileged level and EL3 is the most privileged level. The ISA uses a bit in the PCC to specify whether the hardware thread currently is in Executive or in Restricted mode. To implement the Executive and Restricted mode, the ISA defines new registers as well as new instructions.

[5]This is the description for a points-to-PCC indirect sentry. A points-to-pair indirect sentry has a second capability stored in memory, which is a data capability.

| Conventional (e.g., BR) | Novel (e.g., BRR) |
|---|---|
| E → E | E → E |
| R → R | E → R |
| R → E | |

Table 5.1: Control-flow edges allowed by the conventional and novel instructions specified in the Morello ISA.

The ISA introduces three new registers: RDDC_EL0, RSP_EL0, and RTPIDR_EL0. These three registers are the Restricted mode registers for the DDC, the stack pointer (SP), and the thread pointer ID register (TPIDR). As in other parts of Arm ISAs, e.g., see the implementation of the stack pointer register [14], *banking* is used. That means that on a change from one mode to another the architectural registers are swapped with the contents in the newly introduced registers. In Morello, both Executive and Restricted mode can write to the current architectural register copies of DDC, SP, and TPIDR. However, only executive PCCs can write to the banked registers.

Furthermore, Arm introduces new instructions to constrain how code can change from Executive to Restricted mode and the other way around. Among others, the ISA adds the following instructions: BRR, BLRR, and RETR, which can change from Executive mode to Restricted mode. These three instructions either change from Executive to Restricted mode or the code will stay in Executive mode. These three instructions are modifications of the original Arm instructions: BR, BLR, and RET. The original instructions have a slight change in semantics. The original instructions can do all mode changes, but not from Executive to Restricted mode. Furthermore, all other control-flow instructions in Morello follow the edges laid out by BR, BLR, and RET. The edges are summarised in Table 5.1

Other than by executing a dedicated instruction, e.g., BRR, the only way of performing a transition from Executive to Restricted mode is by entering a capability exception or returning from it.

### 5.2.2 Suitability of Executive-Restricted for supporting library compartmentalisation on CHERI-RISC-V

On Morello, the implementation of library-compartmentalisation is based on Executive-Restricted mode proving its suitability on the Morello platform. However, this does not directly map to CHERI-RISC-V. In the following text, I explain whether the Executive-Restricted mode is suitable for implementation on CHERI-RISC-V.

#### 5.2.2.1 Need for banking

A domain transition in between Executive and Restricted mode does the following operations in one instruction:

- Switch the contents of the DDC register and its banked counterpart register.
- Switch the contents of the SP register and its banked counterpart register.
- Switch the contents of the TPIDR register and its banked counterpart register.
- Jump to the code capability.

In order to facilitate all operations in one instruction, the use of *register banking* is implied. For example, on Arm architectures register banking is used to provide fast access to registers for exception or interrupt handling. However, register banking is not natural to the RISC-V philosophy, which is based on switching registers instead of automatically banking them.

#### 5.2.2.2 Nestability

Other than enforcing the need for register banking, the Executive-Restricted mode has a more fundamental issue. On the same privilege level, only one compartmentalisation mechanism can use the Executive-Restricted mode. Whenever a jump to the Executive mode happens, it has access to the full range of Executive-mode registers. This prevents any more privileged compartmentalisation mechanism to use the Executive mode.

## 5.3 Sufficient set of compartmentalisation primitives

In Sections 5.1 and 5.2, I have shown that the already existing compartmentalisation primitives in the CHERI world are not suitable for supporting library compartmentalisation on CHERI-RISC-V. In this section, I propose an addition to CHERI-RISC-V, that offers a suitable set of compartmentalisation primitives for supporting library compartmentalisation on CHERI-RISC-V.

### 5.3.1 Motivation

One of the results of Section 5.1.3 is that sentries on their own are not sufficient to support library compartmentalisation on CHERI-RISC-V. When using sentries to call into trusted code, the same piece of trusted code can be called from multiple threads and each thread must be able to access its trusted data. Therefore, we identified the need for thread identification when using sentries.

The requirement for reliable thread identification was the outcome of a requirement analysis meeting that the various members of the CHERI research group at the University of Cambridge attended. The contributions below are from me unless explicitly stated otherwise.

In this section, I describe different solutions that allow accessing a per-thread data capability from trusted code. I implement these solutions and evaluate their performance.

### 5.3.2 Thread identification

A straightforward approach is to have a table of per-thread state available, which is only accessible to trusted code. When the trusted code is entered, it needs to find out which entry in the table is the correct one. This can be facilitated via a thread identifier that can be used as an index into the table. This table then can hold the data directly or be a pointer to a further data structure. I will lay out the structure of software access patterns in Figures 5.2 and 5.3.

#### 5.3.2.1 Calling the OS

Retrieving the ID of the currently executing thread can be solved by calling into the entity that provides threading support. In the conventional case, this would mean a call to kernel space at every time trusted code is entered. Using an OS call every time code needs the thread ID would be a performance costly solution.

In order to evaluate this, I measured the cost of a system call on CheriBSD. For this experiment, I used the syscall `thr_self`, which returns the operating systems' notion of the thread identifier[6]. I conducted ten measurements of the cycles taken by the syscall: the arithmetic mean is 718.5 cycles with a standard deviation of approximately 30.41. This measurement methodology and the quantity of measurements is sufficient to get an estimate on the magnitude of how performance expensive regular syscalls are. The goal of these measurements is to determine whether regular syscalls are a sensible way to implement fine-grained compartmentalisation, not to determine the exact cycle number of how long the `thr_self` syscall takes. The measurement code is shown here:

[6]For more information see: `https://man.freebsd.org/cgi/man.cgi?query=thr_self`.

| | Index | Writeable by | Readable by |
|---|---|---|---|
| MTID | 0x780 | {M} + ASR | {M} |
| STID | 0x580 | {M, S} + ASR | {M, S} |
| UTID | 0x480 | {M, S, U} + ASR | {M, S, U} |

Table 5.2: Newly added xTID registers.

```
// Definition of variables
long int tid, stime, etime;
// Start cycle measurement
asm volatile("rdcycle %[dst]" : [dst]"=r" (stime));
// Execute thr_self system call
thr_self(&tid);
// Stop cycle measurement
asm volatile("rdcycle %[dst]" : [dst]"=r" (etime));
```

CHERI compartments enable fine-grained compartmentalisation, which will lead to an increased number of domain transitions. Current library compartmentalisation on Morello manages to do millions of domain transitions per second [90]. Paying the measured cost of around 718 cycles on each transition will have negative performance effect on the overall system. Thus, I conclude from the measurements that there needs to be a different way of both fast and reliably determining the current Thread Identifier (TID) other than calling the operating system.

#### 5.3.2.2 Using the CSR space

**Architectural design**

In this approach, I add a `TID` register per privilege level on the CHERI-RISC-V architecture. I defined these registers in the CSR space. The `xTID` register is a 64 bit wide register and holds the current thread identifier. With this solution, the trusted code does not need to perform a syscall anymore, but can directly read from the CSR register. An `xTID` register can always be read by the current privilege mode and by all more privileged modes. For writing, the code also needs Access System Register (ASR) permission. The ASR bit is a monotonic permission bit in the CHERI capability format. It constrains the access to CSR and Special Capability Register (SCR) registers as described in the CHERI-RISC-V ISA [93]. The newly added CSR registers are summarised in Table 5.2.

The TID is not the same as Thread-Local Storage (TLS). When jumping from one compartment to another, the TLS should change per compartment in order to avoid leaking secrets from one compartment to another. However, the TID will remain the same over all calls as all code executes on the same software thread.

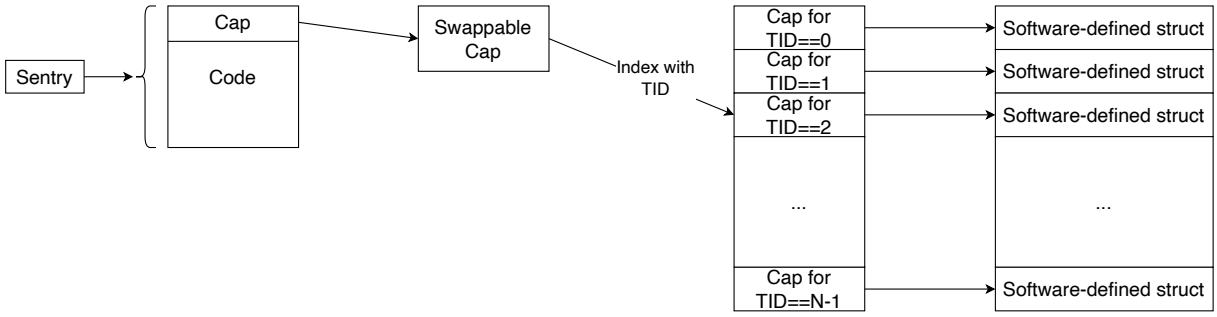



Figure 5.2: Flow of retrieving a trusted data capability with the TID mechanism.

The commonly used RISC-V calling convention defines `x4` as the thread-pointer register. I cannot use `x4` as the TID register because it is a general-purpose register. Therefore, any code – including untrusted compartments – can manipulate the `x4` register. This could trick trusted code into believing that it is executing on a different thread and therefore break security guarantees.

**Usage**

In the following paragraphs, I explain how the `xTID` registers can be used in CHERI software. The envisioned use case is depicted in Figure 5.2.

When a thread enters a sentry, it only finds resources available to every other thread entering this sentry. Other than code, the memory range authorised by a sentry can also hold capabilities, which more privileged code has written before, e.g., at link time. These capabilities can be accessed *PCC-relative* and are protected by the code capability's bounds. Thus, no other code on this privilege level can reach these PCC-relative capabilities.

The PCC-relative capability points to another capability, which is expected to be swapped out occasionally during run time. Swapping out happens when software needs to resize and thus reallocate the table of pointers into the software-defined structs. The software model is to have many sentries representing trusted code. Each sentry has at least one PCC-relative accessible capability pointing to the same *swappable capability*. This enables to swap out the data structure holding the per-thread data capabilities. The *swappable capability* points to a table holding entries for all currently mapped TIDs. This data structure is indexed to with the help of the respective `xTID` register. The data structure for each TID is software defined. I expect the software defined data structures to be dynamic, which requires another level of indirection.

| | SCR Index | Writeable by | Readable by |
|---|---|---|---|
| MTIDC | 0x1b | {M} + ASR | {M} |
| STIDC | 0xb | {M, S} + ASR | {M, S} |
| UTIDC | 0x3 | {M, S, U} + ASR | {M, S, U} |

Table 5.3: Newly added xTIDC registers.

#### 5.3.2.3 Extend TID to a capability

**Architectural design**

As an improvement, I propose to extend the `xTID` registers to capabilities and refer to them as `xTIDC` registers. These registers are in the class of SCRs. Table 5.3 contains an overview of `xTIDC` registers and their access permissions. The access permissions remain the same compared to the `xTID` registers. Extending the `xTID` registers to capabilities introduces the need for sealing. The TID register itself is no secret and can be read by unprivileged code. However, unprivileged code can read the content of the `xTIDC` registers and thus can get access to the capability stored in them. In order to avoid capability leaks, privileged code can seal the capabilities stored in the `xTIDC` registers. Untrusted code can still read the capability, but due to its sealed form untrusted code cannot make use of it, i.e., dereference it. Untrusted code cannot write this capability because it does not run with ASR permission. Privileged code can unseal the capability stored in the `xTIDC` registers via an *unsealer* capability it has been provided with during setup time.

**Usage**

I envision the `xTIDC` registers to be used as laid out in Figure 5.3. When the sentry is entered, it has access to its code, its PCC-relative capabilities and the `xTIDC` registers. In my example, I look at user-space compartmentalisation, which means that only the `UTIDC` register is available. The `UTIDC` register is written by more privileged code handling threading. Thus, the content of the `UTIDC` register is thread specific.

On entering the sentry, the code loads the PCC-relative unsealing capability. Next, it loads the capability from `UTIDC` and unseals it with the unsealing capability. As a final step, the code can then access the software-defined structs in order to retrieve the per-thread data capability.

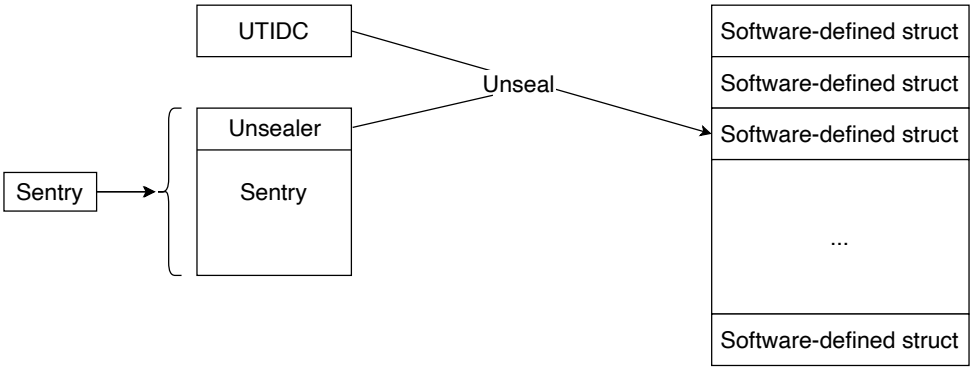



Figure 5.3: Flow of retrieving a trusted data capability with the TIDC mechanism.

### 5.3.3 Implementation

#### 5.3.3.1 TID registers

As a first step, I added the `xTID` registers to the CSR register file in CHERI-Toooba. This step is straightforward and requires the definition of the new CSR indexes as well as connecting a read and write event to the new microarchitectural registers.

While adding the registers, I made the observation that the registers I added differ from most other CSRs. The `xTID` registers do not have any side-effects on the system except when explicitly read or written. This is different to most CSRs, which are used to impact the entire systems state, e.g., writing `satp` to change the root of the address table for paging. A write to a `xTID` register has no effect other than changing the value held by the register and the result of a read to these CSRs is only determined by the value held in the register. Thus, I refer to these registers as *pure-data* CSRs. Apart from the `xTID` registers, I could also identify the *scratch* registers defined in the RISC-V privileged architecture to be pure-data CSR registers.

Handling CSRs in a microarchitecture is not straightforward because CSRs often have system wide impact. Therefore, CHERI-Toooba takes the following approach when encountering a CSR read/write instruction. At register rename, it waits until the ROB is empty. Thus, all older instructions have committed. Next, the CSR instruction is fed to the ALU pipeline where it gets processed. Finally, in the commit stage only one instruction will commit: the CSR instruction. If the instruction included a write to the CSR register file, it will appear in this stage. Furthermore, the core is redirected to the PC following the CSR instruction. This leads to instructions being re-fetched, which is necessitated by some CSR instructions, e.g., a change of the virtual to physical address mapping appeared through a write to the `satp` register. I measured the cost of executing a CSR instruction to be twelve cycles, which is caused by the pipeline drain time. The pipeline is fully emptied before a new instruction is fetched.

For instructions having system effect, this handling is important to ensure that the system operates correctly. However, for pure-data CSR registers, this is not needed because they do

not have an impact on the rest of the system. The `xTID` registers are written on a context switch and are read on a compartment switch. In fine-grained compartmentalisation, the number of compartment switches dominates the number of context switches, which motivated me to optimise pure-data CSR reads in the first instance.

In CHERI-Toooba, renaming waits for the ROB to be empty if the opcode of the instruction is the CSR opcode. I incorporated an additional condition into the renaming logic such that CHERI-Toooba does not wait on pure-data CSR read instructions. This means that a pure-data CSR read instruction is executed in the next superscalar slot, which might be in the same cycle. In the commit stage, I also changed the conditions such that pure-data CSR reads can commit in the next superscalar slot. Therefore, CHERI-Toooba now handles pure-data CSR reads with the same performance as arithmetic operations on general-purpose registers. Writes to CSRs still have to be held back and take a performance penalty.

#### 5.3.3.2 Capability registers

As a next step, I added the `xTIDC` registers to the SCR file in CHERI-Toooba, which consisted of defining the indexes as well as connecting reads and writes to the respective microarchitectural registers. The original CHERI-Toooba implementation already had an improved handling of SCR reads. The SCRs defined in the CHERI-RISC-V ISA are safe to read from the ALU pipeline. They are either pure-data SCRs or they are written on a microarchitectural event that necessitates a *redirect*, e.g., a hardware exception. Thus, I did not implement any additional performance improvements regarding `xTIDC` registers in CHERI-Toooba.

### 5.3.4 Evaluation

I evaluate the minimal compartmentalisation primitives approach on CHERI *library compartmentalisation* [38]. My experiments are inspired by Dapeng Gao's original work on Morello library compartmentalisation[7]. My experiments are microbenchmarks for trampolines in order to measure the cost of a trampoline as precisely as possible. On Morello, there are millions of compartment switches per second motivating to optimise the performance of trampolines [90].

[7]See GitHub repository: `https://github.com/CTSRD-CHERI/cheribsd/blob/dev/libexec/rtld-elf/aarch64/rtld_c18n_asm.S`.

In library compartmentalisation, a trampoline implements the following functionality:

- Stack switching on compartment switch[8].
- Clearing of registers to avoid capability leaks.
- Increasing control-flow guarantees from caller to callee and vice versa.

My test code[9] contains trampolines for all three implementations described in Section 5.3.3. This test code is the basis for the following evaluation.

#### 5.3.4.1 Implementation complexity

**Approach 1: Thread ID as integer approach**

Having the `xTID` registers seems to be the straightforward approach to implement thread identifiers. The registers hold an integer representing a thread ID. However, the fact that trusted code can be entered by any thread necessitates managing a structure for all threads concurrently, and thus it raises complexity for the thread-switching software. When the allocated memory for the software defined structures is not sufficient anymore, the *swappable capability* in Figure 5.2 needs to be changed. In order to maintain correctness, this needs further measures for concurrent programming, e.g., locks or other forms of implementing thread-safe data structures.

The approach using `xTID` registers assumes trust between different threads in the trusted code. In theory, a thread could violate both integrity and confidentiality guarantees. This can be done via writing or reading the data structures indexed by the TID. However, in the compartmentalisation models, the trusted code is supplied by a trusted party and is the same for all threads. Thus, no thread can take malicious action.

**Approach 2: Thread ID as a capability approach**

The complexity of using the `xTIDC` registers is greatly reduced in comparison to using `xTID` registers. The value of an `xTIDC` register is thread-dependent. On a thread switch, the respective `xTIDC` is written with the value of the thread to be scheduled. This can either be done by a thread switcher on the same privilege level or by more privileged code, e.g., the operating system. Having a thread-dependent value in the `xTIDC` registers also avoids the need for thread safety measures.

[8] If this is the first call to a compartment, the trampoline will allocate a stack for the new compartment.

[9] See GitHub repository: `https://github.com/CTSRD-CHERI/RISCV_Comp_Ubench/tree/main/src/libcompart`.

#### 5.3.4.2 Performance in simulation

In order to evaluate the performance of the approaches presented above, I ran the trampoline code on three versions of CHERI-Toooba:

- Implementation of `xTID` registers.
- Implementation of `xTID` registers with fast-reading of *pure-data* CSRs.
- Implementation of `xTIDC` registers.

For the first two implementations, I used the same trampoline code because the fast-reading mechanism is only a microarchitecture optimisation. The code retrieving the trusted data capability is depicted here:

```
cllc ct1, tdc_table_cap // Load address of PCC-relative
                        // capability
clc ct1, 0(ct1)         // Load capability from PCC
clc ct1, 0(ct1)         // Load capability for relocation
csrr t2, UTID           // Get user TID
slli t2, t2, 4          // Shift to 16 byte granularity
cincoffset ct1, ct1, t2 // Calculate address of stack
                        // capability
clc csp, 0(ct1)         // Load stack capability
```

The code first loads the PCC-relative capability which points to a relocation capability. All the sentries will have a static capability pointing to the relocation capability. This enables quick switching between data structures if necessary. Once the data structure has been found it is indexed by the value in `UTID`.

For the `xTIDC` approach the code accessing the trusted data capability looks as follows:

```
cllc ct3, auth_cap     // Get address of authorising
                       // capability
clc ct3, 0(ct3)        // Load authorising capability
cspecialr ct2, UTIDC   // Load sealed capability from user
                       // TIDC
cunseal csp, ct2, ct3 // Unseal data capability
```

Compared to the TID approach, this code does not need to find the root of the data structure because it is already stored in the `UTIDC` register. In order to unseal the content of this register, the code needs to retrieve an authorising capability, which is accessed PCC-relative.

I ran all experiments on the cycle-accurate simulator generated from the CHERI-Toooba Bluespec code. From the simulation output, I gathered the cycles spent in the trampoline.

| | Forward-Trampoline | Backwards-Trampoline | Total |
|---|---|---|---|
| TID | 164 | 146 | 310 |
| Fast-TID | 97 | 77 | 174 (-43.9%) |
| TIDC | 59 | 53 | 112 (-63.9%) |

Table 5.4: Trampoline performance numbers for TID and TIDC approaches.

I used the commit timestamp of the last instruction before the trampoline and subtracted it from the commit timestamp of the last instruction in the trampoline. The measurements are divided into the *Forward-Trampoline* and the *Backwards-Trampoline*, which distinguish the costs making the *call* to a compartment and the *return* to the caller. The non-optimised TID run time is 310 cycles, whereas the fast TID version only takes 174 cycles, which is more than a 43.8% performance improvement. The TIDC trampoline takes 112 cycles amounting to a 63.9% performance improvement over the TID approach. The results are summarised in Table 5.4. These results show the improvements on trampolines, but their impact on overall library compartmentalisation remains future research.

Figures 5.4 and 5.5 show the Konata output for slow and fast TID reading. Konata[10] is a pipeline-stage visualisation tool, which operates on a log file input. During my doctoral research, I have extended CHERI-Toooba to support output for Konata pipeline visualisation.

In Figure 5.4 showing the output of slow TID reading, the CSR instruction remains in the Rename (Rnm) stage until the previous instruction has committed. Furthermore, the instruction succeeding the CSR instruction is re-fetched in the cycle after the CSR reading instruction has committed. All instructions in between are squashed, which is represented in Konata by greying them out.

Furthermore, instructions are greyed out before the CSR instruction marked by the red arrow. This is because CHERI-Toooba fetches for four[11] instruction fragments of which each is 16 bits long. Because all instructions in the microbenchmark are 32 bits long, two 16 bit fragments are merged and thus two Konata executions are greyed out.

Figure 5.5 shows the output of the implementation of fast TID reading in CHERI-Toooba. The CSR instruction commits in the same cycle as the instruction preceding in program order, and the instruction succeeding the CSR instruction in program order commits in the next cycle, which fundamentally improves performance. The commit time-stamps of the preceding and succeeding instructions are only one cycle apart from each other, whereas the time-stamps of these instructions in slow TID reading are 37 cycles apart from each other.

[10]See GitHub repository: `https://github.com/shioyadan/Konata`.

[11]This number comes from multiplying superscalar size times 2. In my setup of CHERI-Toooba, each core is 2-superscalar wide, resulting in four instruction fragments.

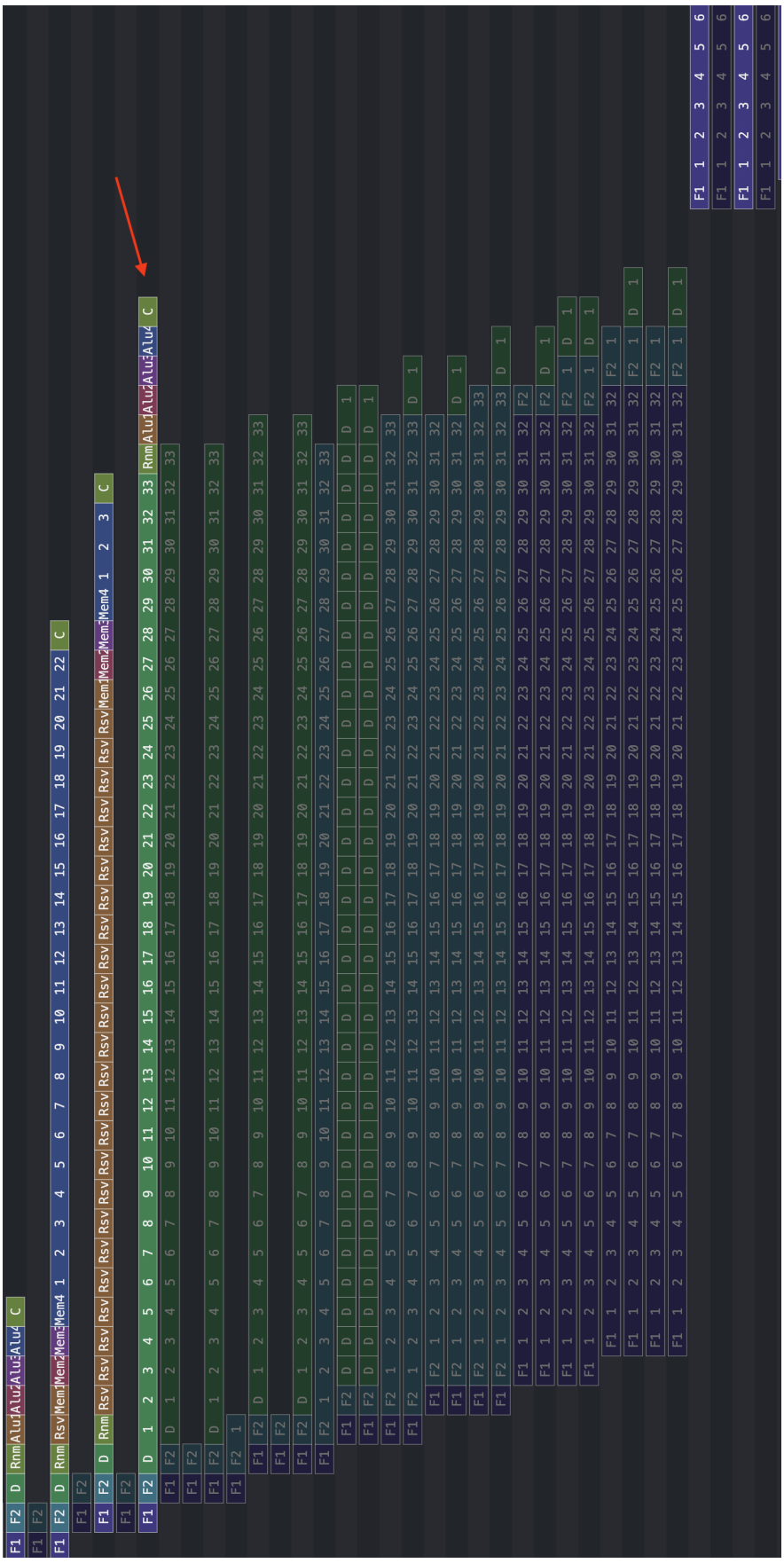


Figure 5.4: Konata output of CHERI-Toooba with slow TID reading. The greyed-out instructions show all the wasted cycles in the original CHERI-Toooba design. The red arrow points to the CSR instruction.

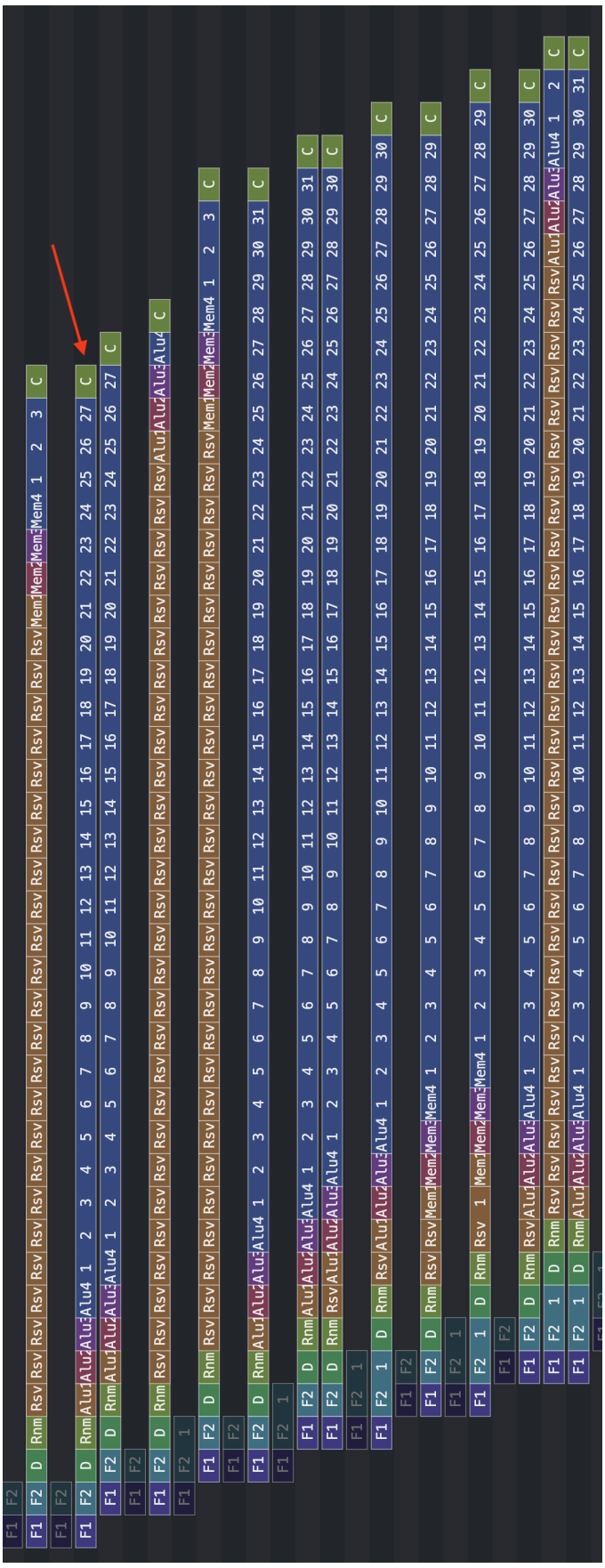


Figure 5.5: Konata output of CHERI-Toooba with fast TID reading. The CSR instruction – pointed to by the red arrow – commits in the same cycle as the preceding instruction.

#### 5.3.4.3 Trampoline performance

In the following text, I summarise the results when testing the TIDC approach running on library compartmentalisation implemented in CheriBSD[12]. The implementation of library compartmentalisation has been conducted by Dapeng Gao and is not a contribution of this thesis.

I measured the cost of a trampoline by calling an empty function in a library. This will invoke a call to the trampoline and then to the target function. The target function will immediately return to the caller through the trampoline. This will give us an approximation of the cost of executing the trampoline code.

My measurements result in an arithmetic mean cost of approximately 82.944 cycles round-trip of the trampoline with a standard deviation of less than 0.443 cycles. Dapeng Gao has carried out measurements of trampoline costs on the Morello implementation, which is based on the *Executive-Restricted* mode. These measurements resulted in approximately 59 cycles per trampoline on average. While the results cannot be directly compared due to being based on different microarchitectures, their difference can be explained. The library-compartmentalisation work on Morello has been a multi-year effort in its bring-up and is based on the *Executive-Restricted* mode. The current implementation of library compartmentalisation is optimised to be the only compartmentalisation model in the user space privilege level because the *Executive-Restricted* mode is not scalable[13]. Compared to the library compartmentalisation implementation on Morello, the work on CHERI-RISC-V was put together in a span of just a few months before submitting this thesis. Furthermore, Morello is an industry-scale implementation that is based on the Neoverse N1 processor with many performance optimisations whereas CHERI-Toooba is a research-scale implementation [4].

These results prove that the TIDC approach can be efficiently used in practice to build compartmentalised CHERI systems. Furthermore, my results provide a motivation for evaluating the TIDC approach on Morello and comparing it against the Executive-Restricted based version of library compartmentalisation.

#### 5.3.4.4 Area

In Figure 5.6, the resource usage overheads of the TIDC design are listed for a single core and the entire SoC on the DE10Pro board passing the standard timing requirements. All numbers show an approximate overhead of 1%, but the usage of registers in the entire SoC case, which is approximately -0.65%. This can be explained by the non-deterministic variables used by the *Quartus* place-and-route tool. The results show that the fast TID and TIDC approaches can be implemented in a modern superscalar, out-of-order machine with little overhead.

[12]See GitHub repository: `https://github.com/CTSRD-CHERI/cheribsd/tree/c18n-riscv`.
[13]For more information, see Section 5.2.1.

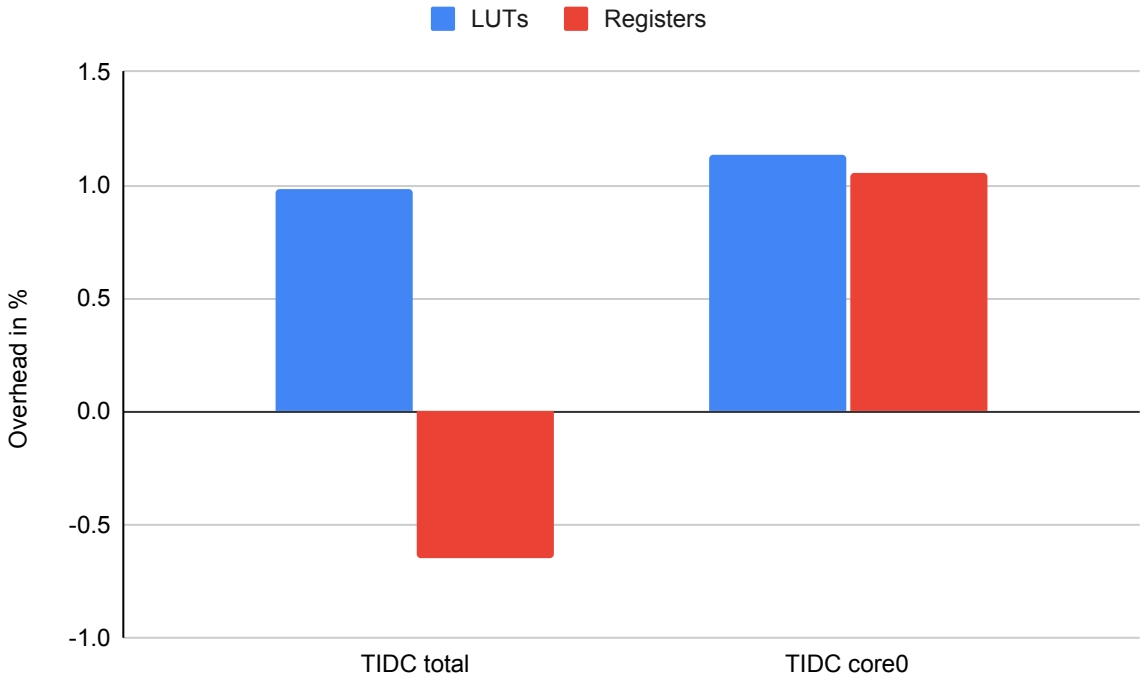



Figure 5.6: LUT and register usage on the DE10Pro board with the CHERI-Toooba design implementing TIDC support. The diagram shows the numbers for the entire SoC as well as for one of the cores.

#### 5.3.4.5 Impact on context switches

The TIDC approach not only touches the userspace, but also the kernel space. In the library compartmentalisation model, a process contains a certain number of compartments. For every process, the content of the `UTIDC` register needs to be different. Therefore, the OS needs to include switching out the `UTIDC` register in every context switch.

For operating systems, the context switch time is important as it has impact on the overall system performance. Therefore, I measure the impact of setting the `UTIDC` on every context switch. I run the context switching benchmark of *unixbench*[14] on two kernels and compare the results. One kernel is enabled for library compartmentalisation and one kernel is on the `dev`[15] branch, which is the branch the library compartmentalisation work is based on. This provides the difference of context switching costs when `UTIDC` needs to be restored.

The results summarised in Figure 5.7 show the unixbench context switching code running on both kernels for 10 seconds. The results demonstrate how many context switches have taken place. On the `dev` kernel I measured approximately 22529 switches taking place, whereas in my measurements the `c18n-riscv` kernel manages to do approximately 22562 context switches in the same time. While these results depict an approximate 0.15%

[14]See GitHub repository: `https://github.com/CTSRD-CHERI/cheri-unixbench/blob/cheri-unixbench/UnixBench/src/context1.c`.
[15]See GitHub repository: `https://github.com/CTSRD-CHERI/cheribsd/tree/dev`.

increase in context switching performance, this needs to be interpreted as a measurement error.

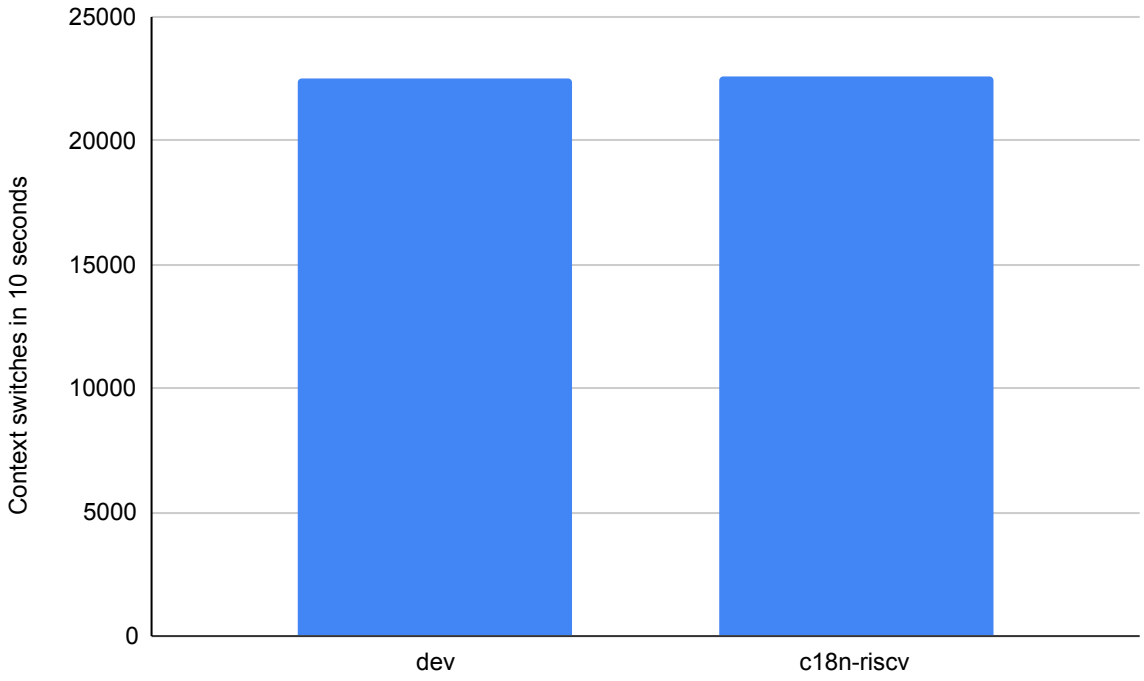



Figure 5.7: Context switching performance numbers of the `dev` kernel compared to the kernel supporting CHERI-RISC-V library compartmentalisation. The extended kernel context switch does not lead to a performance decrease.

While one might expect a performance penalty induced by the slow writing of CSR and SCR registers, this cannot be measured. A kernel context switch is a large and complex code sequence with multiple performance intensive parts, e.g., changing the virtual address space. It is likely that the penalty for changing out the `UTIDC` register is shadowed by one of the performance intensive parts in the kernel context switcher. Therefore, I conclude that my TIDC approach does not impact context switching.

## 5.4 Enhancing domain switching in CHERI-RISC-V

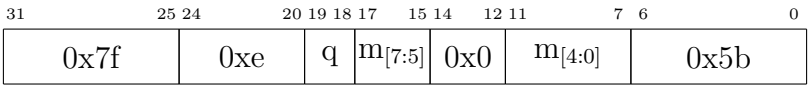
The main purpose of domain switching code is to securely switch from one domain to the other without leaking anything in between. Most importantly, compartmentalisation aims to avoid leaking capabilities between compartments. This commonly includes zeroing out registers in order to avoid information being stored in the registers after the domain crossing code has passed over to the called compartment. The CHERI v9 ISA [93] defines the `CClear` and `FPClear` instructions, which were not implemented in CHERI-Toooba and are not implemented in any of the Cambridge cores implementing CHERI-RISC-V. Furthermore, the Morello ISA [6] does not define a corresponding instruction.

The `CClear` instruction in the CHERI-RISC-V ISA [93] is defined as follows:

**CClear**

**Format**

CClear q(uarter), m(ask)




**Description**

Capability registers $8 \times q + i$ are each set to `NULL` if the ith bit of $m$ is set, with the exception that the 0th bit of $m$ refers to `DDC` when $q$ is 0, rather than `c0`.

### 5.4.1 Design

I propose to change the definition of `CClear` to the following with no other changes to the instruction:

Capability registers $8 \times q + i$ are each set to `NULL` if the ith bit of $m$ is set.

This leads to one bit of the instruction being not used because the `C0` register is defined to always hold the zero capability. However, defining `CClear` as is seems to be unnecessary for CHERI compartmentalisation. `CClear` is an optimisation developed for *purecap* code, which does not make use of the DDC register. Furthermore, defining the `CClear` instruction to touch the DDC register would force microarchitects to make a decision between renaming part of the SCR register file or introducing more complexity for implementing the instruction. If there are no renaming mechanisms for SCR registers, an instruction writing such a register will need to block until it is safe to write that register as elaborated more in Section 5.3.3. In order to avoid enforcing microarchitectural implementation mechanisms, I argue that changing the `CClear` instruction to the definition above is an architectural improvement.

The `FPClear` instruction has a similar functionality, but operates on a window of the floating-point register file. In RISC-V, the floating-point register file does not have a static zero register. Therefore, the `FPClear` instruction does not have a special case defined in the CHERI-RISC-V ISA.

### 5.4.2 Implementation

I implemented both the `CClear` and the `FPClear` instructions in CHERI-Toooba. In order to satisfy the purpose of accelerating domain transitions, executing a clearing instruction should take a single cycle. A straightforward implementation choice would be to write zeroes to all registers selected by the bit mask of the instruction. However, this would mean up to eight writes to the physical register file, which would need a drastic increase of write ports. Instead of pursuing this approach, I decided to change the mapping in the renaming stage whenever a `CClear` or `FPClear` instruction is encountered. The zero register (`x0` or its capability extension `c0`) is a physical register, which is never written to.

In normal renaming, an architectural register is renamed to the physical register that has seen the last write for it. On every write, a new physical register is written.[16] For every register that shall be zero-ed by a clearing instruction, I change the renaming to dynamically point to the zero register instead of pointing to physical register that has seen the last write for the corresponding architectural register. Now, when the architectural register is read, the ALU will read the zero register instead of the last value written. Whenever an architectural write occurs by a younger instruction, my implementation deletes the dynamic mapping and follows the conventional procedure of allocating a new physical register, which will hold the value written.

The implementation of this was not invasive by having a bit vector for each architectural register in the general-purpose and floating pointer register file. The bit vector indicates whether the physical zero register shall be taken or the physical register holding the last written value. This means that `CClear` and `FPClear` set the bits in the corresponding bit vector and then can immediately be marked as `executed` in the ROB and do not need to go through the ALU pipeline.

### 5.4.3 Evaluation

#### 5.4.3.1 Theoretical performance advantage

When implementing `CClear` or `FPClear`, this means that these two instructions can write eight registers at a time per superscalar execution slot. This means that the performance of both instructions will scale with an increase in pipeline width. Figures 5.8 and 5.9 demonstrate the throughput of clearing the entire floating-point register file with conventional `fcvt.s.l f0, x0` instructions compared to `FPClear` instructions. The CHERI-Toooba design used in this research has two superscalar slots for the Fetch front-end, but only one pipeline that is responsible for handling Floating Point Unit (FPU)

[16] Even more, in CHERI-Toooba every instruction is expected to have reserved a physical register for its writeback value. This also holds for instructions with no destination register, e.g., store instructions. I kept this design for implementing `CClear` and `FPClear` and allocated a physical register for them even though it is not needed.


Figure 5.8: Konata output of CHERI-Toooba for clearing the floating-point register file with conventional zeroing instructions (`fcvt`). CHERI-Toooba commits one floating-point instruction per cycle.

instructions. The diagram in Figure 5.8 shows that one FPU instruction commits every cycle as well as the back pressure from having two instructions fetched per cycle in the front-end. This leads to the front-end structures being filled and the front-end not being able to fetch new instructions during some cycles as shown by empty lines in Figure 5.8. Nevertheless, a floating-point instruction commits every cycle amounting to 32 cycles needed to clear the entire floating-point register file. In comparison, Figure 5.9 shows the performance of four `FPClear` instructions that zero out the entire floating-point register file. The instructions are spread over three cycles, but only need four superscalar slots. This means that execution time effectively amounts to two cycles needed. Therefore, the results show an increase of 16 times the number of cycles spent in conventional clearing compared to using `FPClear`.

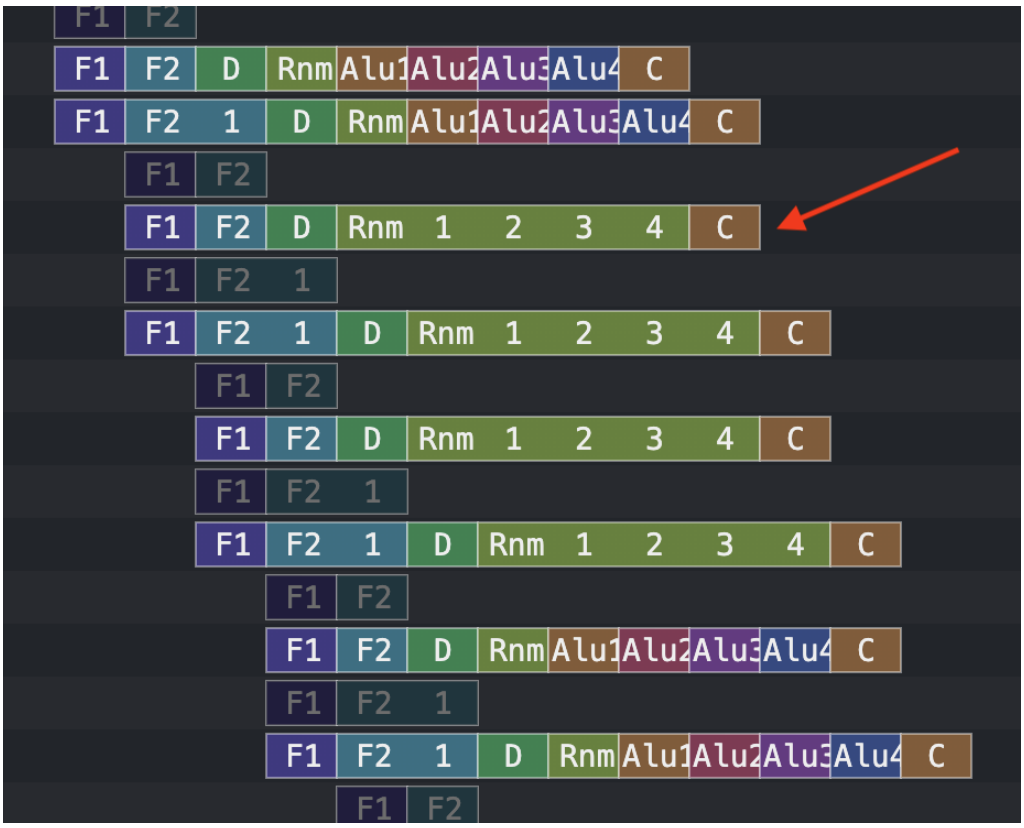



Figure 5.9: Konata output of CHERI-Toooba for clearing the floating-point register file with `FPClear` instructions. The first `FPClear` instruction is marked by a red arrow.

| | Arith. mean | Standard deviation |
|---|---|---|
| 32-fcvt insts | 119.192 | 0.254 |
| 4 FPClear insts | 85.049 | 0.889 |

Table 5.5: Cycle count for clearing the entire floating-point register file with `fcvt` and `FPClear` instructions.

#### 5.4.3.2 Impact of fast floating-point register clearing on trampolines

The trampolines designed by Dapeng Gao currently don't have floating-pointer register clearing. For stronger security guarantees than the current ones, trampolines might need to ensure that no unintended data leaks through the floating-point register file. I applied the experiment from Section 5.4.3.1 and measured the impact of using conventional floating point clearing compared against using `FPClear` instructions. The results are summarised in Table 5.5.

The results show that the `FPClear` variant of the trampoline takes approximately 34 cycles less than the variant with `fcvt`-based clearing. This shows that `FPClear` can have large impact on overall trampoline performance.

| | Arith. mean | Standard deviation |
|---|---|---|
| No clearing | 82.944 | 0.443 |
| 1 reg clearing | 101.977 | 0.605 |
| 1 reg clearing (NOP padded) | 82.923 | 0.566 |
| 2 reg clearing | 82.604 | 0.168 |
| 3 reg clearing | 102.903 | 0.211 |
| 3 reg clearing (NOP padded) | 83.890 | 0.447 |

Table 5.6: Cycle count for clearing different number of argument registers in trampolines with `addi` instructions.

#### 5.4.3.3 Impact of fast general-purpose registers clearing on trampolines

In order to measure the impact of fast clearing on trampolines, I measured first the difference between clearing multiple argument registers. This gives an overview of the performance improvements to be expected. I measured the cycle cost for clearing no, one, two, and three argument registers. The results for clearing no and two argument registers led to comparable results: Not clearing registers takes approximately 82.9 cycles on average, whereas clearing four argument registers takes approximately 82.6 cycles on average. However, when clearing one argument register, I measured approximately 102 cycles execution time for the trampoline. I obtained a similar result for clearing three registers. This can be explained by a shift in code alignment and thus different cache and prefetcher performance properties.

I padded the trampoline clearing one and three argument registers with NOPs which leads to comparable performance numbers. In CHERI-Toooba, NOPs are not treated specifically and take a superscalar slot in the back-end. Thus, the performance results are impacted by the NOPs. The results are summarised in Table 5.6.

When looking at the results of clearing registers, the performance gain of substituting `addi` instructions with `CClear` instructions will be minimal. Thus, I conclude that having fast register clearing for the current form of library compartmentalisation is not worth the investment in additional area and logic.

#### 5.4.3.4 Area

When synthesising the design presented in Section 5.4.2 for the DE10Pro board, the *Quartus* tool gives the resource usage numbers presented in Figure 5.10. The additional microarchitectural storage needed for the mapping of architectural registers to the zero register can be measured in approximately 2.1% LUT overhead for the entire SoC design. The relative overhead for a single core is consequentially larger and amounts to approximately 3.0%. Comparable overheads can be measured for register usage on the DE10Pro.

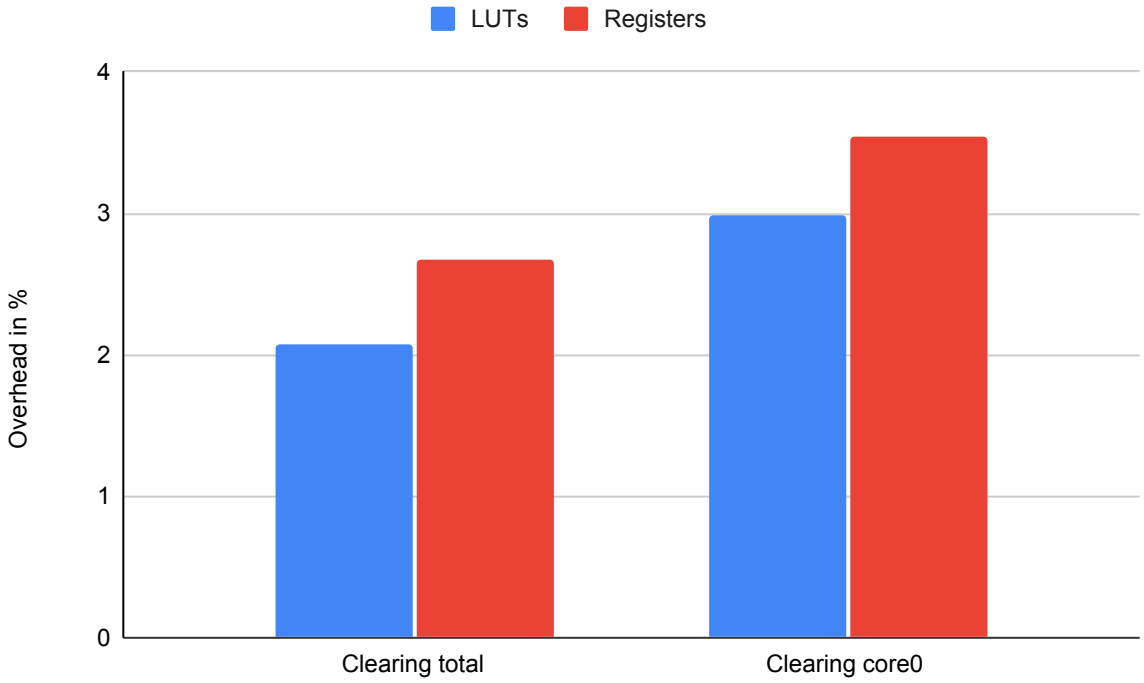



Figure 5.10: LUT and register usage on the DE10Pro board with the CHERI-Toooba design implementing `CClear` and `FPClear`. The diagram shows the numbers for the entire SoC as well as for one of the cores.

#### 5.4.3.5 Discussion

Library compartmentalisation is still in improvement for CHERI-RISC-V and will see performance increases. At the current stage, implementing `CClear` appears not advisable. While the implementation costs are only minor, it is not justified by a measurable performance improvement. However, with a general improvement of library compartmentalisation on CHERI-RISC-V, the impact of fast general-purpose register clearing has to be reviewed.

Implementing `FPClear` had a large and measurable impact on trampoline performance. With FPU pipelines often not having the same pipeline width as the ALU pipeline, the minor cost of implementing `FPClear` is worth the performance benefit.

While library compartmentalisation is currently the most widely used form of compartmentalisation on CHERI-RISC-V, there exist other approaches like co-location[17] or kernel compartmentalisation on CheriBSD [101] in the CHERI world. These need to be evaluated for fast register clearing separately.

[17] See `https://github.com/CTSRD-CHERI/cheripedia/wiki/Colocation-Tutorial`.

## 5.5 Conclusions

While many primitives for lightweight compartmentalisation were already present in the CHERI v9 ISA, they were found to not be sufficient for library compartmentalisation. In this chapter, I evaluate the Executive-Restricted mode mechanism present in Morello, which is used by the pre-existing library compartmentalisation approach on Morello. Motivated by the shortcomings of the Executive-Restricted mode, I design the TID mechanism and explore variants of it, which enables software to securely switch data capabilities between compartments. My results show that the TID approach requires performance improvements within CHERI-Toooba in order to reduce the general overhead of lightweight compartmentalisation. Furthermore, my work concludes that extending the TID register to the capability width is beneficial for both implementation complexity and overall performance. Moreover, my work gives evidence that performance enhancements for fast domain switches can be worth the extra area and implementation cost. I expect my results to generalise well to other microarchitectures and ASIC implementations. Fast access to a subset of the CSR register file appears to be expected for industry-scale RISC-V implementations. Furthermore, my implementation of fast register clearing scales with the ROB size giving reason to expect it to work well within larger implementations.

This chapter proves Hypothesis H.5 by designing and implementing the TID and TIDC approaches. The library-compartmentalisation approach designed by Dapeng Gao successfully runs on CHERI-Toooba.

# Chapter 6

# Protecting microarchitectural state

Transient-execution attacks become a larger threat when they cross domain boundaries. This allows attackers to leak information from the victim domain to their own domain. Cross-compartment transient-execution attacks are mainly facilitated by two microarchitectural techniques:

1. Speculative execution also including out-of-order execution. Executing a younger instruction can be viewed as speculating that the older instruction will commit. This can be extended to pipelined execution. An instruction in the pipeline is executed under the assumption that it will commit, which not always proves to be true.

2. Sharing of microarchitectural information across compartments. Before transient-execution attacks emerged, general-purpose processors were not aware of protection domains, e.g., processor speculation did not distinguish between different processes defined by operating systems. Following this, the microarchitectural state is used by all concurrently existing protection domains.

Without either of the two mechanisms, transient-execution attacks relying on cross-compartment training or leaking of microarchitectural state are not possible. Speculative execution is essential for the performance goals of modern sophisticated microarchitectures. In order to enable high performance goals, but still enable transient-execution attack security guarantees, I have chosen to prevent the sharing of microarchitectural information across compartment boundaries. The research in this chapter has not yet been published in any shape before.

## 6.1 Approach

By default, software shares microarchitectural structures, e.g., on a regular application class processor it is common that all processes time sharing a core are reading from and populating the same microarchitectural structures. From a conventional architectural view, this would not pose a security problem because these microarchitectural structures are not architecturally specified and the result of a program is not dependent on them. However, state from these microarchitectural structures can be made architecturally visible, e.g., through timing side-channel attacks [108].

As laid out in BSC presented in Section 3.6, sharing state between compartments is a threat whereas code in the same compartment can share microarchitectural prediction state. This is because different code segments within a compartment trust each other and have access to all state (data), so it does not make sense for code in a compartment to attack itself via transient-execution attacks. Thus, my approach is to not change prediction within a compartment, but to separate the microarchitectural prediction state of different compartments.

In this chapter, I investigate two approaches to separating microarchitectural structures: tagging and partitioning. It is important to distinguish tagging used in this chapter from CHERI tags. A CHERI tag is a validity tag indicating whether a capability is valid and can be used for memory references. A tag in this chapter is used for identification helping to distinguish between the state of different compartments.

Tagging means that every data unit in the microarchitectural structure is extended by a tag field, which stores a unique value corresponding to the compartment which has written this data. In a simple case, this value could be an integer that is software defined. Whenever the microarchitectural structure is read, the tag value is compared to a value the current compartment has, e.g., in a special purpose register. Only if the two values match, the data access is successful.

Partitioning means that the microarchitectural structures, e.g., prediction state, are split into multiple smaller parts and each compartment is allocated one part. All read and write operations of a compartment only go to its partition and do not interfere with any other partition. Instead of having one big predicting structure for a purpose, the implementation now has multiple smaller prediction structures.

Both approaches have their advantages and drawbacks. The tagging approach allows for more flexibility. If one compartment needs more space in a prediction structure and another one less, the resources will be dynamically available to them. Compared to that, partitioning provides a guarantee of dedicated microarchitectural space. This ensures that there is always a certain minimal number of prediction entries available for each compartment.

The results presented in Section 6.6 indicate that using partitioning and tagging with compartmentalised programs does not lead to a performance overhead. The numbers do not yet suggest a certain partition or tag size. This is caused by the software support for compartmentalisation being in its infancy. The area overhead for more than four partitions does not appear to be tolerable at the current engineering stage. The numbers do not offer conclusive evidence for implementations with a tagged BTB. However, it is important to note that the implementation presented in this chapter is an early prototype and will likely see considerable performance improvements in its next iterations.

## 6.2 Design

This chapter examines the overhead associated with adding partitioned and tagged speculation prediction structures to CHERI-Toooba.

First, I identified the microarchitectural structures I want to protect in CHERI-Toooba:

- Branch direction predictor: this structure returns a prediction in a boolean answer on whether a direct branch[1] is *taken* or *not-taken*.
- Branch target buffer: this structure predicts the target address of an instruction. If the instruction is not a branch, the predictor will give an `Invalid` answer with a high probability[2].
- Return stack buffer: this predictor is specialised on `return` instructions and speculates on the PC value after executing a `return` instruction.
- Store-to-load predictor: this predicts the dependencies of memory operations and determines whether they can speculatively pass each other in order.

These are the four structures that I have identified to perform speculation in CHERI-Toooba. Furthermore, these structures are the main reason for transient-execution attacks [22]. However, this is not an exhaustive list, e.g., store buffers do speculative value forwarding in CHERI-Toooba [104]. While this leaves the possibility of attacks not being covered, previous research could not identify any attack vectors [34, 35]. In the following paragraphs, I explain my approach to separating the information from the different compartments.

[1] A direct branch has an immediate target.

[2] The predictor might still give a valid branch target if another branch collided with the entry to be predicted.

### 6.2.1 Definitions

This chapter introduces Partition and Tag Identifiers (PTIDs). In order to simplify distinguishing between the different identifiers presented in this thesis, I give a short summary of their purpose.

Introduced in Chapter 5, a **Thread Identifier (TID)** is a software-controlled value that identifies a software thread. A TID can have architectural support as implemented via the `xTID` registers. A software thread can execute a compartment, then leave this compartment and enter another one via compartment transition.

In order to distinguish between compartments, systems may employ a **Compartment Identifier (CID)**. This uniquely identifies a software-defined compartment in the current system state. As eluded to above, a thread can be associated with multiple CIDs through its lifetime. Again, an architecture might decide to offer support for this, e.g., via a dedicated register.

A **Partition and Tag Identifier (PTID)** is a microarchitecturally defined value that determines which part of the microarchitecture's resources can be currently used for prediction. In order to change from one partition or tag to another partition or tag, the architecture needs to offer some form of communicating that from software to hardware. For example, this can be facilitated via an architecturally-defined CID. It seems a likely implementation choice to have a limited number of PTIDs in order to save area when implementing the microarchitecture. In order to maintain security, PTIDs likely need to be revoked.

### 6.2.2 Microarchitectural storage of PTIDs

One approach to implementing partitioning and tagging is to save the entire software-defined ID in the tags or use it as a partition identifier. This solution requires many bits in microarchitectures as the software view of an identifier for a compartment is likely to involve a large number of bits. Therefore, I propose the microarchitectural PTID in my design. This is a small bit field that is used as the microarchitectural number for tags and to identify partitions. A mapping is maintained between a large architectural ID and a much smaller microarchitectural PTID. The mapping can be changed to support a working set of compartments. This raises the questions of how many microarchitectural compartments to support at the same time. In Section 6.6, I present my evaluation on what was the best size for the microarchitectural PTID.

In my design, I decided to use a fully associative structure to store CIDs. The different identifiers used throughout this chapter are summarised in Section 6.2.1. Using a non fully associative structure would lead to collisions, which might cause unnecessary flushes and might not make the best use of the structure. Having a limited number of microarchitectural

PTIDs will still cause evictions because multiple software-defined IDs need to be mapped to the same microarchitectural PTID. However, my design still needs to ensure the security property of not being able to read or write any microarchitectural predictor state that does not belong to the currently executing code. For that, I apply the following algorithm:

**Input**: aptid: architectural PTID
table: mapping of architectural to microarchitectural PTIDs
**Output**: mptid: microarchitectural PTID
table: updated mapping of architectural to microarchitectural PTIDs

```
1: aptid ← set_by_some_event
2: if !entryExists(aptid) then
3:     if !full(table) then
4:         mptid ← nextFreeEntry(table)
5:     else
6:         mptid ← getRandomEntry()
7:         shootdown(mptid)
8:     end if
9:     table[mptid] ← newEntry(aptid)
10: end if
```

This algorithm finds a microarchitectural PTID to replace and in line 7 then calls the `shootdown()` method for this microarchitectural PTID.

### 6.2.3 Branch direction predictor

The branch direction predictor in CHERI-Toooba is a tournament predictor featuring a local and global predictor as well as a predictor to choose between the results of the local and global predictors. Partitioning the tournament predictor is straightforward as it only means creating instances of the tournament predictor with a smaller predictor size. However, tagging the tournament predictor is less straightforward because of the global history register. This register would need to be partitioned in order to ensure no cross training or information leakage. For this evaluation, I have only implemented the fully partitioned tournament predictor.

### 6.2.4 Branch target buffer

In CHERI-Toooba, the branch target buffer is an associative cache-like storage structure. This structure can be partitioned straightforwardly by having multiple smaller instantiations of the associative storage structure. The BTB also allows for tagging because the BTB entries are independent and adjacent entries can be occupied by different compartments. I have implemented both a partitioned variant as well as tagged variant of the BTB.

### 6.2.5 Return stack buffer

The Return Stack Buffer (RSB)[3] is a microarchitectural structure that provides predictions for function returns. In CHERI-Toooba, this microarchitectural structure is a ring buffer that stores the most recent return addresses by pushing the next-PC of a `call` instruction onto the ring buffer. When a `return` instruction is encountered, the RSB is popped and control flow is speculatively diverted to this address. Tagging entries in an RSB is not sensible because there is a dependency between them. Two neighbouring entries are one software `call` apart from each other. Therefore, I have not implemented a tagging approach for the RSB in CHERI-Toooba. Partitioning the RSB is done by creating multiple instances of the original module as described before.

### 6.2.6 Store-to-load predictor

The store-to-load predictor shall predict whether a load is dependent on previous stores. In the absence of memory barriers, independent loads can be issued before independent stores and thus save performance. In CHERI-Toooba, this is implemented with an associative storage structure. As a first step, I have only implemented partitioning to accelerate the development time for a working prototype.

### 6.2.7 Shooting down compartments

Shooting down a microarchitectural PTID means erasing all its state, which is necessary to avoid any information leakage between compartments. It is essential that the shootdown has finished before the new compartment with the same microarchitectural PTID runs any instruction. Therefore, an ongoing shootdown needs to block all instructions from the new compartment until the shootdown is completed. The shootdown time needs to be minimised, which is explained in Section 6.3.3.

An overview of the system designed is shown in Figure 6.1.

## 6.3 Implementation

In this section, I describe the implementation of my PTID design presented in the previous section.

[3]In CHERI-Toooba, the module implementing this functionality is called Return Address Stack (RAS). In order to maintain consistency, I have kept the term Return Stack Buffer(RSB) throughout this thesis.

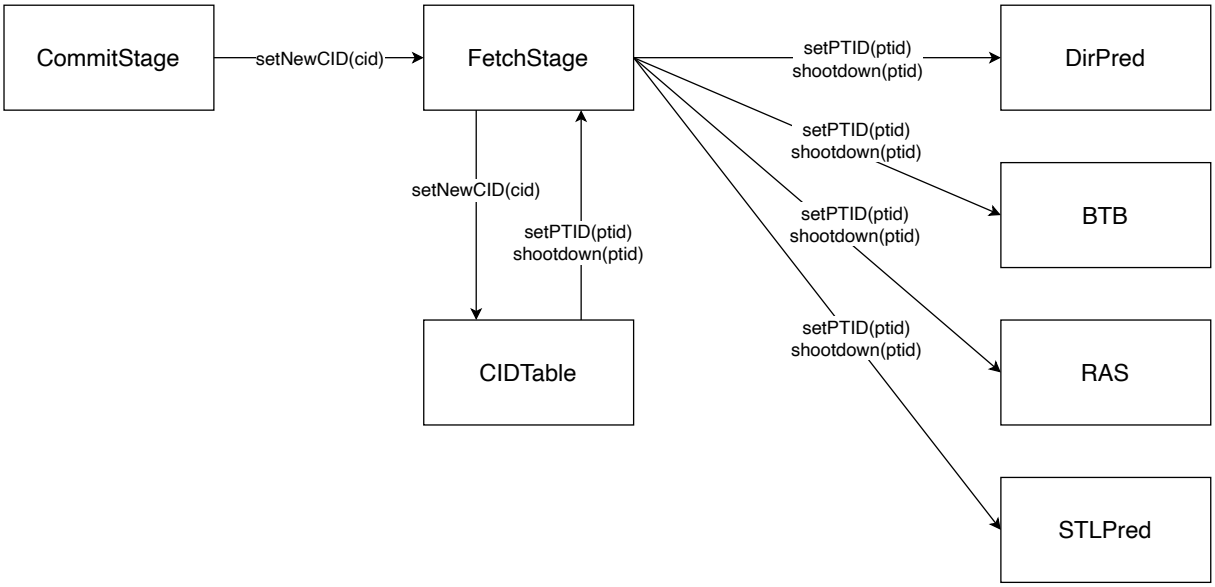



Figure 6.1: This diagram shows my design for supporting partitioning and tagging prediction state in CHERI-Toooba. All prediction structures exist as a partitioned variant. The BTB also exists as a tagged variant.

### 6.3.1 Partitioning state

Partitioning state can be implemented straightforwardly in CHERI-Toooba. I took the existing modules and created wrappers as well as necessary interfaces to them. These wrappers create multiple instances of the prediction module with each instance being dedicated to one PTID. The wrappers then determine at run time which of the actual prediction modules to call based on the current PTID. The choice of partition is dynamically changed by an architectural compartment switch as described in Section 6.4.

When creating multiple instances of the same prediction module, I was confronted with the choice of leaving each instance to be the same size, i.e., the same number of entries, or to divide the number of entries by the number of instances. I chose to divide all resources for prediction units, but the RSB. The standard depth of the RSB in CHERI-Toooba is 8 entries. Splitting this would lead to a significantly limited call-depth speculation capacity. The implications of this on the FPGA resources used by my design are evaluated in Section 6.6.2.

### 6.3.2 Tagging the BTB

Tagging the BTB with the current PTID works by extending the entry by an additional tag. This tag contains the value of the PTID. The entire BTB is shared between all compartments and every compartment can write every entry thus allowing for flexibility in how much BTB space is currently needed by one compartment. However, the BTB will only respond with a target if the PTID of the currently active compartment matches the value in the tag. The extended BTB entry is depicted in Figure 6.2. The tag value stores

the current PTID information. Furthermore, the entry contains a validity bit indicated by *V*. I added this to enable invalidating entries within one cycle, which allows implementation of a fast shootdown mechanism as presented in Section 6.3.3.


Figure 6.2: BTB entry in my extended CHERI-Toooba design. The parts marked in orange are extensions to the BTB entry existing in baseline CHERI-Toooba. The validity bit is denoted by the character *V*.

### 6.3.3 Fast shootdown mechanisms

As laid out before in Section 6.2.7, shooting down a microarchitectural compartment must be a blocking operation. In order to minimise the overhead of separating microarchitectural state, I have implemented shooting down as a single-cycle operation in all prediction structures.

In CHERI-Toooba, most microarchitectural structures for storing data are implemented using some form of a Block Random Access Memory (BRAM) module on an FPGA. This also holds for all prediction modules. A BRAM has a maximum number of read and write ports, which limits the data that can be written per cycle. In order to zero out a BRAM, one needs `BRAM_SIZE/NUM_WRITE_PORTs` cycles. With the number of write ports on a BRAM usually not exceeding 2, the number of stall cycles encountered becomes quickly unacceptable[4].

In order to mitigate this penalty, I introduce a validity bit for every entry in a prediction structure. For example, in BTBs this leads to an additional bit in the entry as depicted in Figure 6.2. The validity bits can be implemented as a bit vector stored in data flip-flops, which can be written in a single cycle. Only prediction entries with the validity bit set can be used for speculation. Every write to a prediction structure will set the validity bit, but also overwrite the content of the entry and therefore this design prevents any leakage.

The fast shootdown mechanism presented here is tuned for FPGA implementations, which offer BRAM. ASIC designs do not rely on BRAM, but they rather use SRAM (Static RAM). The implementation presented in this section can be equally mapped to SRAM. However, it is important to note that ASIC designs are highly customised and thus likely will find a better solution than the one presented above.

[4]In common cases on CHERI-Toooba, this would range from multiple hundreds to sometimes even multiple thousands of cycles.

## 6.4 Choice of communicating compartment switches

With the design implemented, the remaining task is to communicate a change of compartment to the microarchitecture. There are many design options of which an additional one is explored later in this dissertation (see Section 7.3). A straightforward and less complex design approach is to introduce a special-purpose register. In my experiments, I have implemented the (Partition and Tag ID) `PTID` register, which I have allocated at index `0x4a0` in the RISC-V CSR space. When this register is written, the microarchitecture switches to the new PTID. This is a user readable and writeable register, which means that user space has full control over what the microarchitecture interprets as a compartment. This is sufficient for prototyping microarchitectural design solutions, but future research is needed in order to adapt this mechanism to current threat models[5].

Using a special-purpose register to indicate a compartment change requires two new operations before the domain switch can happen: determining the PTID for the next compartment and setting the register to this value. Determining the PTID for the target code will include a load from memory. The code is depicted in the following snippet:

```
// Before goes other trampoline code
cld t0, offset(MEM_CAP)   // Load software ID from memory
csrw ptid, t0             // Write ID to PTID register
cjalr cra, TARGET_SENTRY // Jump to destination compartment
```

The CSR write leads to a microarchitectural barrier. The performance implications of this have been evaluated in Section 5.3.

## 6.5 Security evaluation

Separating branch predictor state mitigates all transient-execution attacks that rely on training a predictor from a different domain or leaking data from it. This includes the domain crossing Spectre-BTB attacks. My tagging and partitioning work does not protect from intra-domain transient-execution attacks. However, in my threat model that does not pose a violation. A compartment is narrowed down to code that can trust each other. If there was still a risk of code attacking each other in a compartment, the instance of the compartment would have been wrongly chosen. The threat can be mitigated by further compartmentalising the code.

While the approach presented in this chapter offers protection during running a compartment, this guarantee does not necessarily hold when a transition between compartments is done. At transition time, there will be instructions from both the calling and the called

[5]Using the TID presented in Chapter 5 is not sufficient because there exist multiple compartments per Thread ID, which makes the TID too coarse-grained.

compartment in the microarchitecture. This offers still room for exploitation and needs to be addressed. I approach this separately in Chapter 7.

Furthermore, the approach in this chapter does not mitigate transient-execution attacks that rely on training prediction structures within a compartment, e.g., a Spectre-PHT attack that works by mistraining a branch direction predictor via architecturally allowed values. This transient-execution attack path needs to be mitigated via other means, e.g., by extending the architectural speculation contracts presented in Chapter 3.

## 6.6 Performance and area evaluation

In order to evaluate the suitability of this approach to real-world programs, I have run the `nginx`[6] webserver program on CHERI[7]. In order to test my PTID design, I used the library-compartmentalisation approach on CHERI-RISC-V. For this, `nginx` is well-suited because it has six libraries linked against the main program: `libcrypt`, `libssl`, `libcrypto`, `libc`, `libthr`, and `libsys`. The SPECint2006 benchmark did not appear to be a good choice because it is compute-heavy and I therefore do not anticipate many domain transitions. For more information on the library compartmentalisation and its supporting primitives on CHERI-RISC-V, see Chapter 5.

### 6.6.1 Running library compartmentalisation on CHERI-RISC-V

The library-compartmentalisation software implementation on CHERI-RISC-V is currently in a prototype status and will see improvements in the future. One limitation of the current implementation is the software corpus that can be run. Library compartmentalisation separates the stacks of different compartments from each other.

In many cases, argument passing for C functions is done via argument registers. However, in some cases the stack needs to be used for argument passing:

- A function has more arguments than the underlying architecture has argument registers.
- A function makes use of `varargs`.

In case of RISC-V, the architecture has eight argument registers[8]. This means that functions with nine or more arguments will use the stack for argument passing on conventional RISC-V systems.

[6] `https://nginx.org/`.

[7] The CHERI team maintains a branch of `nginx`(GitHub repository: `https://github.com/CTSRD-CHERI/nginx`) that has fixes to run on CHERI. I took this branch and made slight modifications to enable to build `nginx` on CHERI-RISC-V.

[8] In the common ABI, these are `a0` - `a7`, which correspond to `x10` - `x17`.

| | Cycle overhead for stack switching |
|---|---|
| 1 library | 2.33% |
| 2 libraries | 2.53% |
| 3 libraries | 2.47% |
| 4 libraries | 2.40% |

Table 6.1: Trampoline cycle overheads when doing stack switching compared to a non-stack switching variant of the trampoline. This table entails calling 1, 2, 3, and 4 libraries.

With functions from different libraries no longer sharing stacks in library compartmentalisation, passing an argument via the stack becomes impossible. In the library-compartmentalisation implementation on Morello, this was addressed by creating a new ABI. A similar approach has to be carried out on CHERI-RISC-V. Due to the tight timing of bringing up library compartmentalisation on CHERI-RISC-V, the new ABI has not yet been implemented.

In order to still run real-world software on library compartmentalisation, I asked Dapeng Gao for a version of his approach that has all functionalities of library compartmentalisation, but does not perform stack switching. Between library function calls, trampolines are still invoked, but they do not switch over the stack in order to produce working code.

In order to evaluate the performance improvement introduced by doing no stack switching, I have implemented a small benchmark that calls multiple different library functions, which immediately return. This gives an estimate of the overhead of stack switching for the current state of the library compartmentalisation on CHERI-RISC-V. The results are summarised in Table 6.1.

The results show that the cycle overheads remain the same over all run benchmarks. These numbers can be explained by the additional instructions needed for calculating the offset of the stack to be switched in. In my benchmarks, the same trampoline was called over and over again leading to good caching results. My experiments approximate a smaller working set of compartments. I would expect a workload with a large number of compartments to suffer from limited cache size and thus degrade in trampoline performance.

### 6.6.2 Area

In the following text, I evaluate the resource usage of the CHERI-Toooba design for partitioning and tagging support. I evaluate the design for instantiations with 2, 4, and 8 partitions. Furthermore, I estimate the overhead of tagging the BTB. The results for resources overhead on the SoC- and core-level are presented in Figure 6.3. One can see two trends in Figure 6.3. First, the overhead for 2 partitions is small. The LUT numbers do not differ and can be interpreted as noise, and the register overhead remains within one percent for the overall design. Second, the greater the number of partitions, the more the overhead of registers used goes down and even into a negative overhead for the eight-partition design. This is likely an artefact of the synthesis tool and needs to be addressed in further iterations of the design.

In Figure 6.4, the results are depicted for resources overhead for the designs with a tagged BTB. The numbers are a result of the non-optimised prototype design and show high overheads in either LUTs registers or both.

I deem that both for tagging as well as for partitioning that the implementation overhead can be improved. My implementation approach was targeted towards fast prototyping and chose to duplicate structures with smaller sizes. This likely leads to more BRAMs being instantiated which then have a higher probability to be underfilled. A more sophisticated implementation strategy might be able to avoid this and optimise BRAM usage.

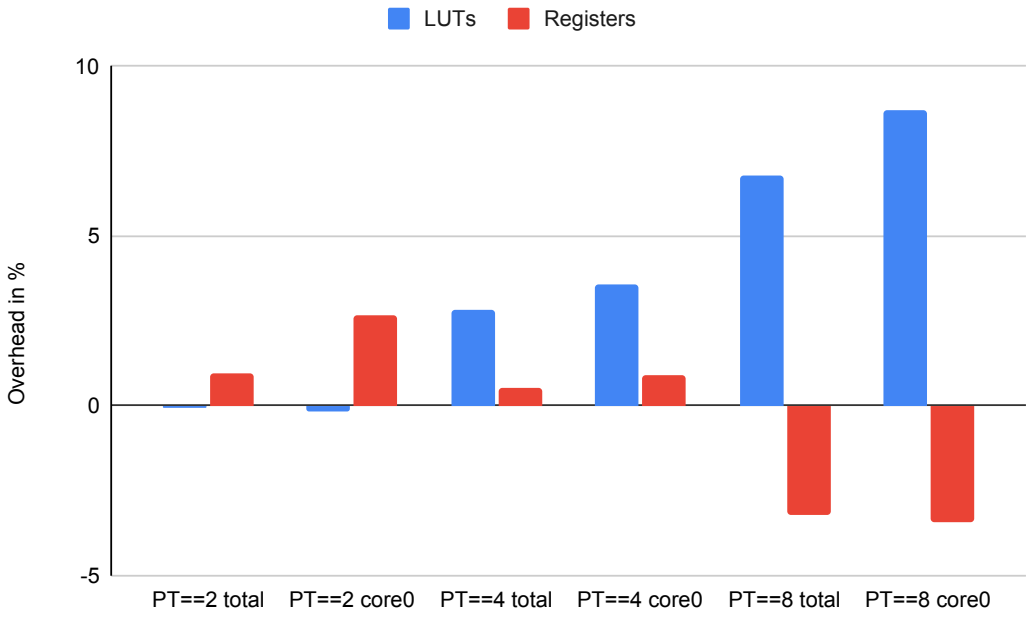



Figure 6.3: LUT and register usage in the CHERI-Toooba design implementing support for partitioned microarchitectural structures. This diagram shows configurations with 2, 4, and 8 microarchitectural compartments. For each configuration, the diagram shows the numbers for the entire SoC as well as for one of the cores.

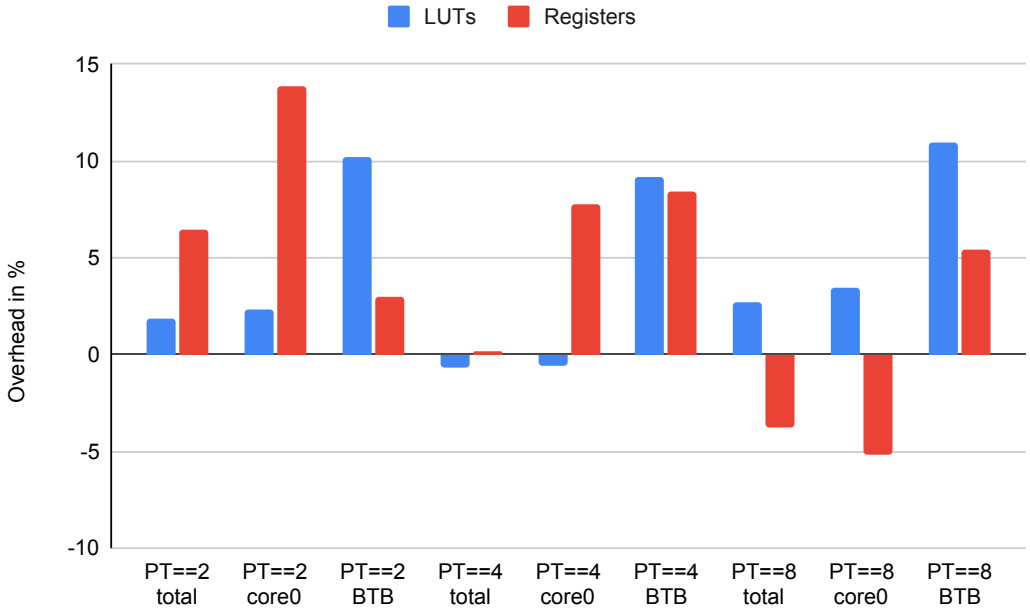



Figure 6.4: LUT and register usage in the CHERI-Toooba design implementing support for partitioned microarchitectural structures with exception of the BTB, which is tagged. This diagram shows configurations with 2, 4, and 8 microarchitectural compartments. For each configuration, the diagram shows the numbers for the entire SoC, for one of the cores, and for the BTB.

| | no-comp | comp | comp with uarch PTIDs |
|---|---|---|---|
| PTNumber=2 | 137.27s | 137.86s | 137.77s |
| PTNumber=4 | 137.32s | 138.25s | 136.67s |
| PTNumber=8 | 136.66s | 137.98s | 137.46s |

Table 6.2: Timing measurements of running the `fetchbench` benchmark on CHERI-Toooba with the aforementioned microarchitectural prediction structures being partitioned. This table lists the numbers for the no-compartmentalised, compartmentalised, and compartmentalised cases with 2, 4, and 8 microarchitectural compartments.

### 6.6.3 Performance

With the approach presented in Section 6.4, the PTID is set via a CSR write. As seen with the `xTID` approach, writing to a CSR is performance costly in CHERI-Toooba. In this setup, I need to write the `PTID` register four times during a full trampoline invocation:

1. Set the PTID to the trampoline's value.
2. Set the PTID to the callee's value.
3. Set the PTID to the trampoline's value.
4. Set the PTID to the caller's value.

I measured the cost of the non stack-switching trampoline instrumented with the code added that sets the `PTID` register four times. In the arithmetic mean, the trampoline takes approximately 42 cycles longer when instrumented to carry out PTID switching. This was measured with a standard deviation of approximately 0.247.

#### 6.6.3.1 Performance with all structures being partitioned

In order to test partitioning of microarchitectural prediction structures, I have run the `fetchbench` benchmark[9] on configurations of CHERI-Toooba with space for 2, 4, and 8 microarchitectural compartments. This `fetchbench` issues 200 fetches of the same page split over 5 parallel processes, leading to 1000 requests in total. The results of these experiments are summarised in Table 6.2.

The numbers show that there is no performance difference in any of the configurations. It seems likely that the `nginx` run time is dominated by other costs. It remains future work to test for applications that are more dominated by the compartment switching cost.

[9]See GitHub repository: `https://github.com/CTSRD-CHERI/nginx/blob/master/fetchbench`.

| | no-comp | comp | comp with uarch PTIDs |
|---|---|---|---|
| PTNumber=2 | 137.32s | 137.92s | 137.17s |
| PTNumber=4 | 137.53s | 137.84s | 137.45s |
| PTNumber=8 | 137.08s | 138.32s | 138.28s |

Table 6.3: Timing measurements of running the `fetchbench` benchmark on CHERI-Toooba with all microarchitectural prediction structures being compartmentalised, but the BTB, which is tagged. This table lists the numbers for the no-compartmentalised, compartmentalised, and compartmentalised cases with 2, 4, and 8 microarchitectural compartments.

#### 6.6.3.2 Performance with the BTB being tagged

I ran the `fetchbench` benchmark in the same configuration on CHERI-Toooba with all microarchitectural prediction units being partitioned apart from the BTB, which is tagged. I used configurations of CHERI-Toooba with 2, 4, and 8 microarchitectural compartments. The numbers are summarised in Table 6.3.

The numbers in Table 6.3 show the same trend as the numbers in Table 6.2. Partitioning and tagging does not have an impact on the performance of `nginx` with the `fetchbench` benchmark.

## 6.7 Conclusions

Compartmentalisation is an important architectural mechanism to reach privilege separation. As with other architectural mechanisms, compartmentalisation needs to be protected against transient-execution attacks. In this chapter, I have shown that a superscalar, out-of-order microarchitecture can implement support for partitioning and tagging of microarchitectural prediction structures. Microarchitectural compartments are defined by writing to a newly added CSR register.

While the work presented in this chapter requires future work in order to become mature enough for real-world implementations, it supports Hypothesis H.6 in parts. Hardware-defined compartments can be implemented as shown by my extension of CHERI-Toooba. Furthermore, I find that partitioning and tagging does not need to have a performance impact on real-world applications. The `nginx` results obtained in this research suggest that its performance is not limited by the number of typical compartment transitions executed. However, my work remains a prototype and does not yet optimise for PPA, which remains future work. An performance improvement is presented in Chapter 7. Because of the early prototype nature of my implementation, I cannot say how my approach would scale to other microarchitectures or ASIC implementations.

The results of this chapter support Hypothesis H.7. With both tagging and partitioning, the microarchitectural structures are separated from each other. This leads to my design providing confidentiality and integrity for CHERI hardware compartments.

# Chapter 7

# Securing domain crossings

In addition to preventing cross-compartment transient-execution attacks while a compartment is running, it is equally important to prevent transient-execution attacks during a domain transition. When switching from one compartment to another, the processor will execute instructions from both the calling compartment and the called compartment.

In this chapter, I discuss the attack scenarios and discuss, implement, and evaluate multiple solutions. The research in this chapter will be published in parts in version 10 of the CHERI ISA technical report.

## 7.1 Transient-execution attack scenarios

A sophisticated microarchitecture executes many instructions at the same time. With a raised number of compartments, the number of compartment switches also increases. Given this, compartment switches need to be fast in order to prevent a high performance impact. During a compartment switch, microarchitectures will execute instructions of two compartments at the same time. The back-end will process the instructions of the old compartment whereas instructions of the new compartment are already in the front-end of the processor. This means that instructions from different compartments can impact the microarchitectural state at the same time, which in turn can lead to successful transient-execution attacks. I have identified the following attack scenarios:

1. *Training prediction state of the other compartment*[1]. An example attack would be to poison a BTB entry of the other compartment such that the other compartment speculatively jumps to the attacker compartment during the next execution.

2. *Leaking prediction state of the other compartment*[1]. A potential attack could be that the attacker measures the run time of its own code, whose run time is determined by the prediction state of the victim. For example, the callee's first instruction can be

[1]Going in both directions: from the caller to the callee and the other way around.

a direct jump whose branch direction lookup collides with an entry from the caller. This can then be used to leak the content of the caller's prediction entry.

3. *Leaking the register file of the calling compartment.* Due to misspeculation, a compartment can be forced to speculatively enter a different compartment without having cleaned up its secret state. The attacker compartment can read out the register state and leak the victim's secrets. An example is described by Shivakumar et al. [86] in which they demonstrate an attack that manages to speculatively exit an AES function and thus reveal an insufficiently encrypted ciphertext.

4. *Speculating to the other compartment*[1]. The attacker compartment can pick a branch target of the victim compartment and jump there. For example, the prediction entry of an indirect branch in the callee's code can alias with the entry of an indirect branch from the caller therefore leading to speculative jumps to the caller.

The attack vectors are depicted in Figure 7.1. All of these attack scenarios need to be mitigated in a secure compartmentalised system. At the current state of research, I think these attack vectors comprise the techniques used by the transient-execution attacks that are most pressing to mitigate. In the remaining parts of this chapter, I present and evaluate different approaches to mitigate transient-execution attacks on domain crossings.

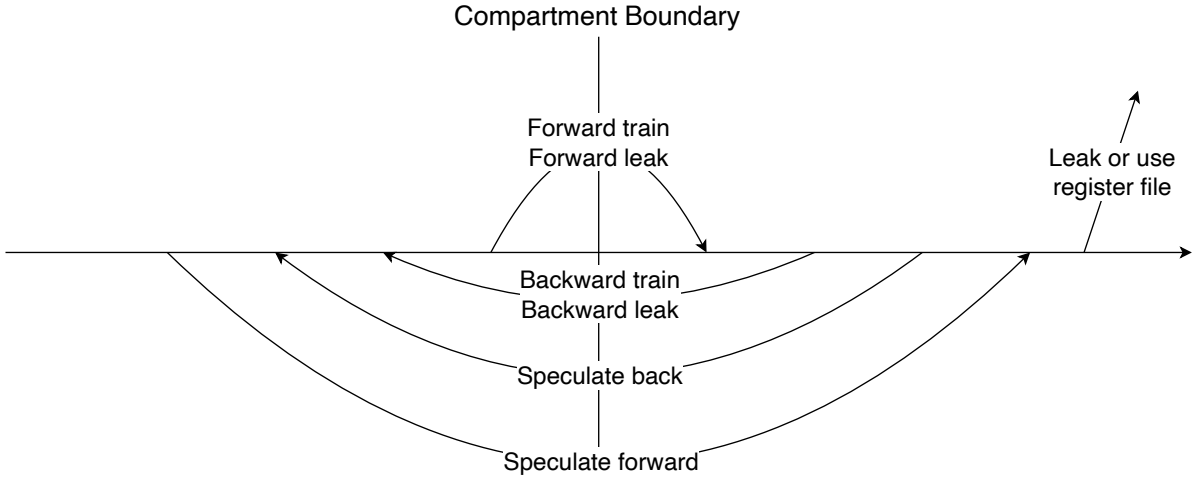



Figure 7.1: Transient-execution attack scenarios on domain crossings. All of these need to be mitigated in order to have a secure system overall.

## 7.2 Fence and barrier instructions

A straight-forward solution is to insert a fence or barrier instruction whenever a compartment switch is executed. A fence or barrier instruction will ensure that no speculation can go into either direction. However, the disadvantages are at hand: executing a fence instruction will completely empty the pipeline and thus likely lead to a performance penalty. Using fences for an effective transient-execution mitigation assumes that the speculation

contracts from Chapters 3 and 4 are in place. Without the contracts in place, an attacker can use all attack avenues closed by the contracts and is not forced to attack domain crossings. In other words, this means that securing domain crossings is only sensible with other mitigation mechanisms like contracts as it would be wasted effort otherwise.

### 7.2.1 Performance impact on CHERI-Toooba

The partitioning and tagging approach presented in Section 6.3 is based on setting a CSR register in order to facilitate a compartment switch. Writing a CSR register leads to a pipeline flush and effectively becomes a barrier instruction. For more information on the implementation of CSR writes see Section 5.3.3. When evaluating the trampoline performance on the approach presented in Section 6.3, each trampoline invocation needs approximately 42 cycles more than a non-instrumentalised invocation. This lays out the penalty of using a CSR instruction as a barrier in CHERI-Toooba.

### 7.2.2 Performance impact on Arm-based ASICs

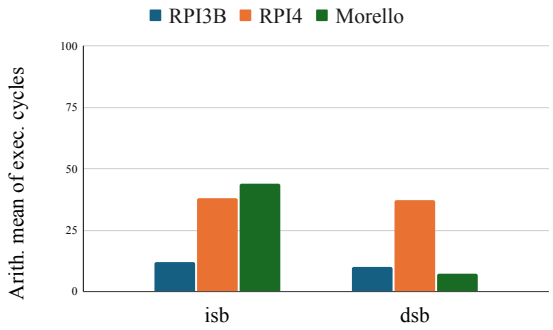
The CHERI-Toooba pipeline does not fully exhibit performance-enhancing features of state-of-the-art microarchitectures. In order to evaluate the potential performance hit on a real-world system, I implemented two microbenchmarks containing barrier instructions and evaluated them on multiple Arm-v8.x implementations.

The first benchmark is the canonical recursive implementation of calculating the Fibonacci numbers. For every call of the function, I inserted one of the following three instructions:

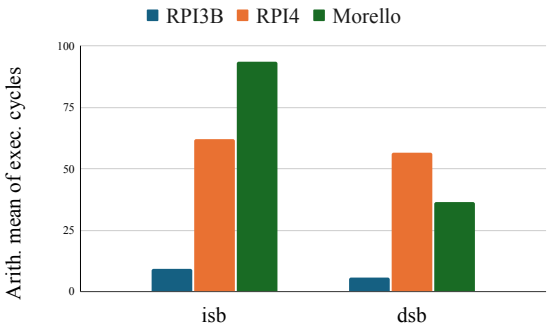
- An `isb` (instruction synchronisation barrier) instruction.
- A `dsb` (data synchronisation barrier) instruction.
- A `nop` (no-operation) instruction.

The second benchmark is a handwritten function which attempts to maximise the current number of instructions in-flight in a microarchitecture. The goal was to maximise the instruction-level parallelism for which I mixed together ALU, FPU, and memory instructions. This experiment was conducted to get a closer approximation of the upper bound for cycle penalties of these instructions because it would cause a full pipeline flush. The upper bound likely is larger than the numbers retrieved by my experiments. Like before, I instrument every iteration with one of the three instructions from above. I ran both benchmarks in all configurations on multiple SoCs with Arm-v8.x based cores:

- Raspberry Pi 3B board with the Broadcom BCM2837B0 SoC with a quad-core Arm Cortex-A53 processor.




(a) Cycle overhead of a Fibonacci program, which has both `isb` and `dsb` instructions inserted. The program was executed on the Raspberry Pi 3B, Raspberry Pi 4, and the Morello board.




(b) Number of cycles spent on `isb` and `dsb` instructions measured in the context of a specifically hand-crafted program that aims to fill up the pipeline. The numbers show `isb` and `dsb` instruction instrumented runs over the baseline execution. The program was executed on the Raspberry Pi 3B, Raspberry Pi 4, and the Morello board.

Figure 7.2: Cost of barrier instructions on Arm architecture-based machines.

- Raspberry Pi 4 board with the Broadcom BCM2711 SoC with a quad-core Arm Cortex-A72 processor.
- The Morello board with a CHERI CPU based on the Arm Neoverse N1 platform with four cores.[2]

Figure 7.2 shows the overheads of the instrumented programs on all three Arm architecture-based machines. The data in Figure 7.2(a) show the results of running `fib(23)`, which will lead to 92735 invocations of the function. The data shows that the penalty overhead for the Fibonacci program in the `isb` case becomes higher the more sophisticated the microarchitecture becomes. The Arm Cortex-A53 processor in the Raspberry Pi 3 is an in-order machine whereas the other two processors tested are out-of-order machines [12, 13, 41]. Another observation is that for the two Cortex implementations, the penalty for the `dsb` case is a few cycles below the `isb` case penalty. However, for Morello, the `dsb` case penalty is significantly lower. This could be explained with Neoverse N1 being a server-class processor, which is optimised for many highly-threaded workloads, which need fast `dsb` performance.

The data in Figure 7.2(b) show the results of running the pipeline filling program, which executed $100,000$ times. As in Figure 7.2(a), the `isb` case clearly shows that the cycle overhead per execution of the function becomes worse with increasing microarchitectural complexity. Similarly, the `dsb` case penalty on Morello is low compared to the `isb` penalty. When looking at the cycle results of the Cortex-A53 it becomes apparent that the penalty for both the `isb` and the `dsb` case does not increase in when running the *fill-pipeline*

[2]The Morello-CPU is described in more detail in Section 2.13.

benchmark. An explanation can be the in-order pipeline, which does not need to flush and wait as much as in the case of the more sophisticated microarchitectures.

When comparing both diagrams, it becomes apparent that fence and barrier instructions are very costly on high-end microarchitectures. In Figure 7.2(b), the arithmetic mean of cycles lost was just below 94 cycles. Looking back at the results in Section 5.3.4, executing a library-compartmentalisation trampoline on Morello takes approximately 59 cycles[3], which means that the penalty on for an `isb` is larger than executing an entire library-compartmentalisation trampoline. Therefore, using a barrier to stop execution is not a viable path forward to securing domain crossings against transient-execution attacks.

## 7.3 Compartment ID sealing

Protecting compartments against transient-execution attacks is crucial for the overall security of CHERI systems. Code and data associated with a compartment must be somehow identified and validated when they are being accessed. One approach is to give compartments identification numbers. Giving protection domains an identification number has traditionally been done in software, e.g., processes are assigned Process Identifiers (PIDs) in operating systems. In some cases, a software ID can correspond to a hardware ID, e.g., PIDs can correspond to Address Space Identifiers (ASIDs). However, that is not necessarily the case. In this proposal, I suggest an architectural ID for compartments. I call these numbers Compartment Identifiers (CIDs).

### 7.3.1 Approach

I propose to import CIDs into the capability format. Having this solution leads to an atomic change of CID and code capability – when the code capability is changed, the CID is also changed. The capability format with the new CID field is depicted in Figure 7.3. In this example, I chose the field to be 10 bits wide, but any choice that fits in the capability format is a valid instantiation of the CID sealing approach.

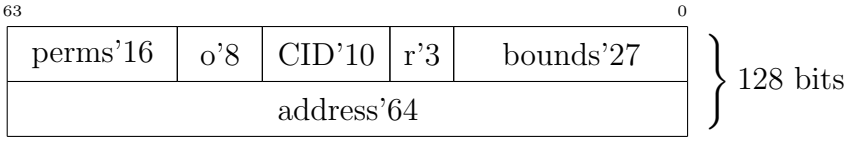



Figure 7.3: Bit representation of a 128-bit capability (excluding the one bit validity tag) with the new CID field included. The otype field is denoted by 'o'. The reserved bits are denoted by 'r'.

All values but the zero CID (CID==0) are valid CIDs. Furthermore, every CID-sealed code capability already is a sealed entry capability. This leads to the fact that every

[3]The measurements of trampoline costs on Morello have been carried out by Dapeng Gao.

capability within a compartment can be manipulated by the compartment itself. This also includes CID-sealed code capabilities.

At the moment, this CID sealing proposal is limited to the 128-bit capability format. However, the underlying mechanism to add CID bits to the meta information bits of the capability works for every format. The CIDs in the capability format are referred to as the Architectural Compartment Identifiers (ACIDs). The number of bits for the ACIDs is known as the ACID_LENGTH. Software may choose to virtualise these and can create Software Compartment Identifiers (SCIDs), which are a concept similar to PIDs. An ACID may consist of the corresponding ACID added to other bits in order to create a virtual identifier. Equally, software can choose to create a SCID that is fully independent of the ACID it is mapped to.

### 7.3.2 Implicit sealing

The current CID (the CID of the current compartment) is determined by the CID in the PCC, which is also a capability. If the PCC changes and the new PCC has a different CID, this constitutes a compartment change.

With an ACID being in all capabilities, one can establish a concept referred to as CID sealing. All capabilities are implicitly sealed by their CID. Capabilities are considered unsealed if and only if: their CID matches `PCC.CID` or their CID is 0. All other capabilities are implicitly sealed. An implicitly sealed capability can be inspected (its fields can be read), but it cannot be manipulated nor can it be used to access memory.

CID sealing allows a register file to hold capabilities from different compartments without allowing capability leaks[4]. The currently executing compartment can only make use of its own capabilities or explicitly unsealed capabilities.

### 7.3.3 Compartment change

A compartment change is done when the CID is changed. The CID can be changed by installing a code capability with a different CID into the PCC register. Installing a new PCC can be facilitated in multiple ways in CHERI-RISC-V:

- `CJALR`: Capability Jump And Link Register.
- `CInvoke`: jump to a sealed code capability and unseal a data capability with a matching `otype`.
- `mret` and `sret`: return from machine mode or supervisor mode exceptions.
- Any instruction that produces an exception.

[4]This does not prevent information leaks if software uses a capability register to store secret information.

### 7.3.4 Sealing capabilities

I propose to use non-memory capabilities to seal a capability with a CID. I propose to create capabilities that authorise another capability to be sealed with a CID being in a certain range of CIDs. This authorising capability has the same fields as a conventional CHERI memory capability, but uses its fields differently. The address field is interpreted as the CID, and the bounds define a range of CIDs. This allows for code to be granted a range of CIDs it can use to seal other capabilities with.

### 7.3.5 Sharing

Sealing implicitly forbids sharing capabilities between compartments. However, sometimes this behaviour is desired by software. I have designed two mechanisms for sharing capabilities between compartments:

- Explicit unsealing: this will explicitly unseal a capability, and it can be shared either via the register file or via memory. Another compartment can pick up this capability and seal it again and use it. This does not protect against transitive capability leaks - an unsealed capability can be passed on to a third compartment and thus be leaked.
- CID spaces: CIDs can be separated into CID spaces. One approach is to allow unsealing within a CID space. This allows all compartments in that space to use that capability, but no other compartment can unseal the capability. This protects from transitive capability leaks outside of the CID space.

Furthermore, I also envision the possibility of *resealing* capabilities. A capability belonging to one compartment can be resealed to another compartment. This effectively means that code can seal one or more of its own capabilities for another compartment. The main advantages are that this avoids capabilities being unsealed in the open as done with explicit unsealing as well as potential performance improvements avoiding the need to unseal and seal capabilities. However, resealing is a disadvantage for the receiving compartment since it cannot know whether a capability was sealed by itself or another compartment. Therefore, this approach might incur additional validation checks in software.

### 7.3.6 Explicit unsealing

Explicit unsealing is a necessary feature in order to make capabilities available if software desires to do so. A capability with the zero ACID can be CID sealed by any compartment. Therefore, I propose two new instructions:

- `CCIDUnseal cd, cs1`
  If cs1.CID and PCC.CID match, then cs1 will be assigned to cd with cd.CID=0. Otherwise, clear the tag of cs1 and assign it to cd.
- `CCIDSeal cd, cs1`
  If cs1.CID==0, then cs1 will be assigned cd with cd.CID=PCC.CID. Otherwise, clear the tag of cs1 and assign it to cd.

### 7.3.7 Virtualisation and revocation

When employing more compartments than storable in the ACID space, software will need to virtualise CIDs. This can be done in two ways: ensuring unique ACIDs in every address or revoking ACIDs. The latter will lead to one or more ACIDs needing to be reused. In order to maintain safety and security, every capability from the old compartment needs to be made inaccessible. This prevents leaking capabilities from the old to the new compartment where both of them have the same ACID, but different SCIDs.

## 7.4 Mitigating transient-execution attacks

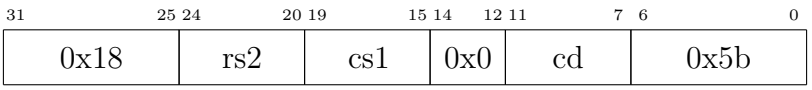
The CID sealing approach is a contribution of this dissertation. Due to time reasons as well as capacity reasons in the CHERI software team, it was not possible to carry out a full software evaluation. Therefore, I implemented a subset of the approach above in order to evaluate how to mitigate transient-execution attacks with CID sealing:

- Integrated the CID field into the capability format. ACID_LENGTH = 10 bits and OTYPE_WIDTH was shrunk to 8 bits.
- Compare the CID of the data capability against the PCC's CID for each memory operation.
- Implement a `CSetCID cd, cs1, s2` instruction: this sets the CID field of `cs1` with the lower bits of `s2` and writes the result to `cd`.

This prototype instruction allows me evaluate the CID sealing approach with a limited effort. In this approach, setting the CID is not secure. However, this is not a limitation of the approach, but only a simplification for rapid development. The CIDs can be set in a secure way as laid out above in Section 7.3.4. Here is a preliminary definition of the `CSetCID` instruction:

**Format**

CSetCID cd, cs1, rs2



### 7.4.1 Implementation of CID sealing

My implementation of CID sealing is a straightforward extension of CHERI-Toooba. I added a new field to all structs in the CHERI capability library[5] in order to represent CIDs and shrunk the `otype` field to make space in the capability format. Furthermore, I added getter and setter methods for the CID field to all representations of CHERI capabilities. The CHERI capability library has multiple representations in order to optimise for the different use cases at different stages in the pipeline [82, 83]. I implemented the changes in all different representations.

In order to implement implicit sealing, I extended the `capChecksMem` function that is called in the memory pipeline. All instructions going through the memory pipeline are checked against CID sealing and throw an exception if the following condition does hold: `(getCID(auth) != 0) && (getCID(auth) != getCID(pcc))`.

I implemented the `CSetCID` instruction in the style of other capability-modifying instructions. I extended the `capModify` function called in the ALU pipeline for this case. In the case of `CSetCID`, the function changes the CID in the capability, but leaves the capability unchanged otherwise.

### 7.4.2 Modifications to enable partitioning and tagging of microarchitectural prediction structures

Instead of using a dedicated CSR register to signal a compartment change, the CID in the PCC register changing determines a compartment switch. While the underlying approach to separating information from different compartments in microarchitectural prediction structures remains unmodified, the method of changing between them has changed. In the approach described in Section 6.4, the microarchitecture has one global CID which is used by all state. In this approach, changing the CID is therefore a global event.

When using CID sealing, multiple CIDs may be used in the microarchitecture at the same time. For example, instructions of a new PCC might already be in the front-end, whereas the back-end still handles instructions of the previous PCC. In order to accommodate this requirement, I developed the following design. This design is depicted in Figure 7.4.

The `CIDTable` now offers a `translate` method, which reads the current mapping of ACIDs and returns either a valid PTID or an `Invalid` signal. The `translate` method is called by all prediction structures whenever a prediction is called[6]. Depending on whether the translation led to a valid PTID, the prediction structure then responds with the corresponding prediction results or returns a *do-not-care* result otherwise.

[5] See GitHub repository: `https://github.com/CTSRD-CHERI/cheri-cap-lib`.

[6] Some of the prediction structures in CHERI-Toooba are multi-cycle predictors and thus require internal storage of the PTID and whether this is a valid value.

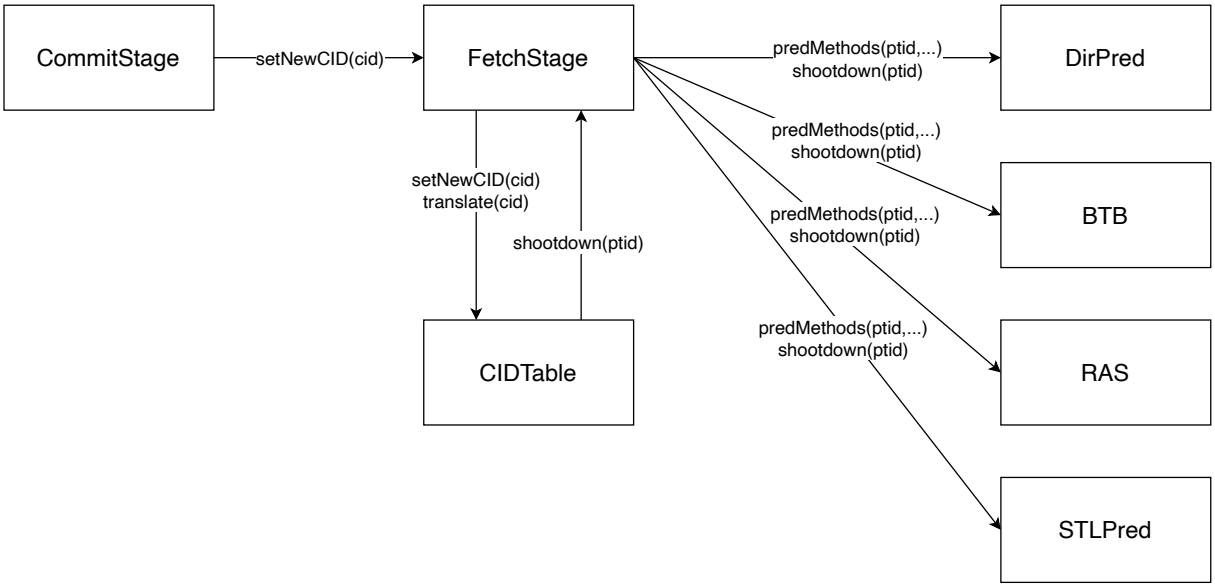



Figure 7.4: Connecting the design for partitioning and tagging branch predictor state with CID sealing. In comparison to Figure 6.1, a new `translate` method is implemented, and all prediction methods now have the PTID as an argument.

I chose to update the `CIDTable` from one writer only, which is the Commit stage. For this prototype implementation, I have only implemented a compartment change on `CJALR` instead of all possible CID changing instructions presented in Section 7.3.3. In my implementation, whenever a valid `CJALR` instruction is encountered, the table is updated. This might include shooting down the compartment that currently occupies the shot down PTID. With this design, only architecturally accessed[7] compartments impact the state of the table of CIDs.

## 7.5 Security evaluation

My CID sealing approach aims to mitigate the security threats pointed out in Section 7.1. I could not test these guarantees in practice because software has not yet been ported for CID sealing. CID sealing leads to some form of *instruction tagging*. Every instruction is connected to a PCC, which holds a CID. Thus, in the pipeline, it can be determined which microarchitectural state an instruction can access.

When predicting to an instruction with a different CID, the predicted instructions will be tagged with the new CID and thus are only allowed to access state of the new CID. In order for this to hold, speculation needs to use always link the target address with the respective CID of the PCC. Under this assumption, instructions and CIDs are thus atomically connected and no confusion can occur.

[7] An architectural jump to a compartment defines access here.

This property disallows any training of a microarchitectural prediction structure or leaking state from it as pointed out in Section 7.1. The instructions can only ever access their own structures.

Furthermore, a speculative jump to a new compartment not only makes the microarchitectural structures of the callee inaccessible, but also the capabilities in the register file. Thus a called compartment cannot leak any capabilities from the caller. However, the callee can still read the register file itself, which means that the callee can leak secrets stored in the register file. If this scenario lies within the threat model, further measures need to be taken by the calling code.

## 7.6 Performance evaluation

In this section, I evaluate my CID sealing approach for performance. For this, I research the impact of CID sealing on library-compartmentalisation trampolines.

In library compartmentalisation, I adapted the trampoline setup code so that every target capability is assigned a CID. I use the `CSetCID` instruction and use a unique value defined in the library-compartmentalisation approach as the CID. I compare the run time of this code to the run time of library compartmentalisation without having CID sealing instrumented jumps. When the trampoline is invoked, the CID-sealed code experiences an approximate 0.14 cycle improvement per trampoline invocation, which has to be classed as noise. Therefore, the CID sealing approach does not exhibit any performance penalty. This is a significant performance improvement over the CSR writing approach presented in Section 6.4, which adds a 42 cycle penalty per trampoline round trip.

The measurements above were carried out without the CID checks enforced in hardware. I made this decision because the software for this approach has not yet been brought up. In CHERI, many data capabilities are produced via the `AUIPCC` instruction and might then be passed on to other compartments. The `AUIPCC` instruction will seal a data capability with the CID of the current PCC, which will prevent other compartments from unsealing it.

## 7.7 Area and resource usage

Implementing the CID sealing approach does not add any additional bits in the CHERI capability format, but rather re-purposes them. Furthermore, it adds one instruction as well as a small amount of logic comparing whether capability memory operations may go ahead. This explains the LUT and register results presented in Figure 7.5. I measured no increase in LUT usage on the DE10Pro and an overall 2.37% overhead in register usage, when synthesising this design.

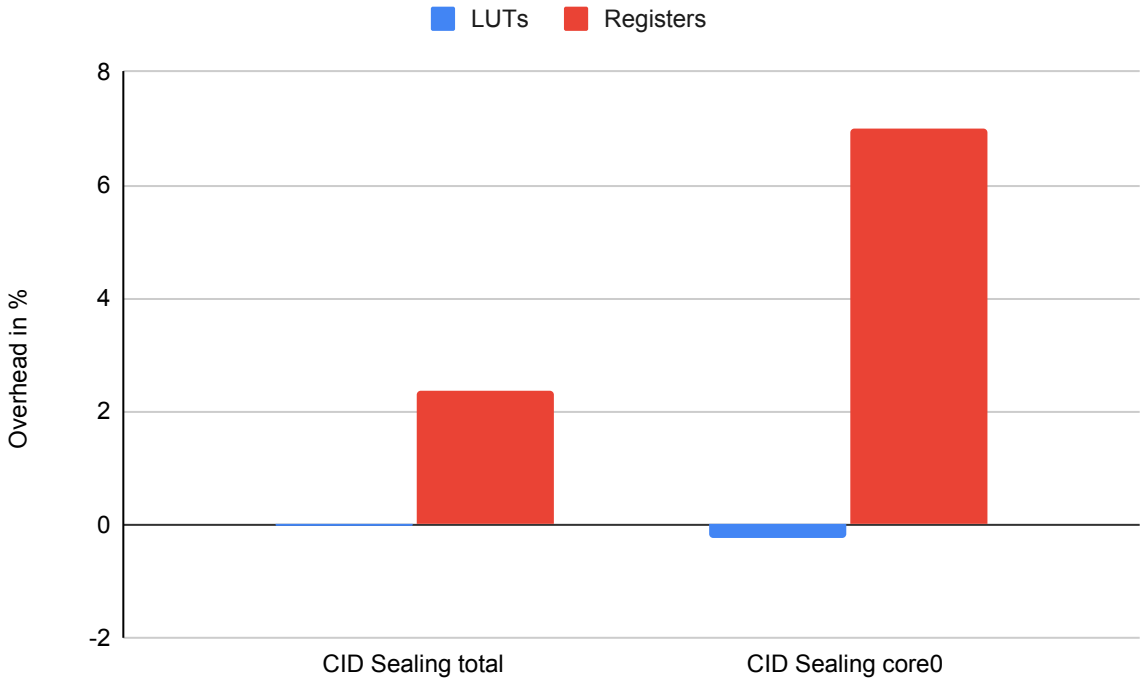



Figure 7.5: LUT and register usage in the CHERI-Toooba design implementing CID sealing.

## 7.8 Conclusions

Transient-execution attack mitigations are required during compartment transitions and within compartments. My analysis of barrier instructions on Arm-v8.x implementations shows that their use can become performance costly. On the Morello microarchitecture, the cost of executing an instruction synchronisation barrier (`isb`) instruction can amount to up to 94 cycles, which is greater than the cost of executing a library-compartmentalisation trampoline. In order to increase the performance of secure domain crossings, I present CID sealing, which is an modification of the CHERI capability format that allows for implicit sealing of capabilities. In my evaluation, I show that CID sealing can mitigate the transient-execution attack scenarios identified before. Furthermore, CID sealing adds no overhead in performance and only a small overhead in resources used for CHERI-Toooba implemented on FPGA. I expect these results to generalise well to other microarchitectures and ASIC implementations because my approach does modifications of the already existing capability format and only requires minor additions to the design, which can likely be carried out in parallel to other tasks in the microarchitecture.

The results of this chapter support Hypothesis H.6. CID sealing adds information on the current compartment to the capability format and thus enables hardware to implement microarchitectural compartments. The performance overheads are not measurable in the CID sealing implementation and the overhead in resources used on FPGA is small.

However, the CID sealing design connected to the partitioning and tagging work presented in Chapter 6 has not yet been tested.

Furthermore, this chapter supports Hypothesis H.7. With CID sealing the attack scenarios described in Section 7.1 can be mitigated.

# Chapter 8

# Conclusions

In this chapter I present high-level conclusions spanning all of the research results presented in this dissertation followed by a more detailed analysis of the evidence supporting each hypothesis. Furthermore, I point out future work opportunities.

## 8.1 High-level conclusions

Transient-execution attacks are complex to assemble and it is difficult to reason about them because they trigger an unspecified hardware behaviour from software. This specification vacuum led to various software and hardware approaches towards transient-execution attack mitigations, which were independent of the other side. In my work, I have advanced the state-of-the-art in reasoning about transient-execution attacks and provided primitives to assemble mitigation mechanisms. The contracts for general-purpose architectures are the first ones to my knowledge that manage to reason about speculative execution and significantly constrain the attack interface.

My research has shown that in order to mitigate transient-execution attacks, hardware and software need to carefully interact. This is essential in order to maintain the performance design requirements of modern hardware and software. On the hardware side, this includes a full audit for each microarchitecture. Microarchitects need to explore which kind of state their microarchitecture has and which must not be shared between protection domains. This will then lead to architectural mechanisms to prevent information from leaking from one compartment to the other.

CHERI capabilities are pointers that are enriched by metadata developed to address memory safety vulnerabilities in computer systems. These metadata are helpful for mitigating transient-execution attacks. As demonstrated in previous work, the bounds of CHERI capabilities allow natural mitigation of Spectre-PHT attacks [34, 35]. In this work, I have designed architectural speculation contracts for general-purpose architectures. I have extended this work through speculation contracts for CHERI systems. The capability

metadata[1] is used to constrain which possibilities speculation has. This is a convenient use of architectural capabilities, which are already present for architectural memory safety. Conventional systems that want to have fine-grained constraints on the possibilities of speculation need to add additional architectural and microarchitectural state to achieve this goal, e.g., as demonstrated by Yu et al. [109].

CHERI Compartmentalisation has been shown to have great architectural security benefits [38]. However, CHERI compartmentalisation also needs to be protected against transient-execution attacks. In my work, I show that CHERI compartments can also be hardened to maintain confidentiality and integrity in the face of transient-execution attacks. One of the main benefits of CHERI compartmentalisation compared to other forms of compartmentalisation is the speed of domain transitions [38]. In my work, I present Compartment Identifier (CID) sealing, which manages single-cycle microarchitectural domain switches and offers transient-execution mitigation.

Overall, although the problem of mitigating transient-execution attacks on CHERI systems is not yet fully solved, I have provided the framework necessary to reason about transient-execution attacks on CHERI systems. In this work, I have shown that CHERI's various architectural security primitives can be extended to enable similar microarchitectural security guarantees against transient-execution attacks while retaining performance goals.

## 8.2 Answering hypotheses

This section answers the hypotheses from Section 1.2 by accumulating the evidence shown throughput this work.

**Hypothesis H.1**

Conventional processor architectures can be extended with architectural speculation contracts to constrain overall speculation.

While previous work showed the potential of architectural speculation contracts, its practical application was lacking. In Chapter 3, I present three contracts: Translation Speculation Contract (TSC), Branching Speculation Contract (BSC), and Memory Ordering Speculation Contract (MOSC). I carefully designed them through multiple iteration stages, which converged in the prose specification of these contracts. I then applied these contracts to the RISC-V architecture and implemented three generators in the QuickCheckVEngine framework to test RISC-V implementations against my contracts. The generators discovered a set of violations in CHERI-Toooba that could result in transient-execution attack scenarios, e.g., speculative diversion of control-flow outside the control-flow possibilities allowed by the contracts. Thus, Hypothesis H.1 is supported by these results. My contracts

[1] Most importantly bounds, hardware permissions, and sealing information.

manage to mitigate many serious transient-execution cross-training scenarios. A future formal analysis of information leakage needs to prove whether my contracts are exhaustive at preventing transient-execution attacks.

**Hypothesis H.2**

A conventional microarchitecture can be extended to implement the speculation contracts and does not incur significant performance overheads.

Implementing architectural speculation contracts in a way that respects design constraints and performance goals is important to prove their feasibility in real-world systems. In Chapter 3, I implemented the changes needed for CHERI-Toooba to adhere to the Branching Speculation Contract (BSC). The violations were that CHERI-Toooba allowed training in speculation, which then led to non-committed targets ending up in the BTB and RSB structures. Fixing this, required training to be delayed until the target was branching-non-speculative. In the arithmetic mean, the results showed that my contracts can lead to a 1.8% performance overhead, with some of the benchmarks even leading to a performance improvements. Therefore, this thesis proves Hypothesis H.2.

**Hypothesis H.3**

The CHERI architecture can be extended using architectural speculation contracts that constrain the behaviour of microarchitecture in speculation in order to avoid information leakage.

CHERI introduces a large number of additional architectural states when mapped into a real-world architecture. The latest release of the RISC-V CHERI architecture contains more than 280 pages[2]. The feasibility of speculation contracts needs to be also proven on CHERI architectures. In Chapter 4, I designed the Capability Speculation Contract (CSC) targeting the abstract CHERI architecture. The contract is dependent on the definition of CHERI capabilities, but not on the exact specification of instructions in the concrete instantiation of the CHERI architecture. The CSC forbids attack vectors, which have been identified by previous research [34]. The automatic testing method generated counterexamples for all known violations of the Meltdown-CF class. My work presented in Chapter 4 supports Hypothesis H.3. However, I have not conducted an exhaustive list of leakage possibilities, which remains future work.

**Hypothesis H.4**

A CHERI microarchitecture can be extended to implement the architectural speculation contracts and does not incur significant Performance, Power, and Area (PPA) overheads.

[2] `https://github.com/riscv/riscv-cheri/releases/tag/v0.9.0-prerelease`.

Implementing CHERI in modern architectures leads to overheads. When implementing speculation contracts on CHERI, it is crucial to not further increase these overheads. In Chapter 4, I presented implementations that fix both data-CSC and implementation-CSC violations. When changing the behaviour from *faulting* to *clearing* when encountering an exception, both area numbers as well as performance numbers remained unchanged. This proves that data-CSC can be implemented without significantly affecting the current processor design. The SinglePCC design led to smaller overheads than the Morello design and saved area compared to the standard CHERI-Toooba design proving that instruction-CSC can be efficiently implemented. This shows that CHERI implementations can adhere to speculation contracts, therefore proving Hypothesis H.4.

**Hypothesis H.5**

CHERI-RISC-V can support library compartmentalisation, previously only available on the Morello platform.

Library compartmentalisation is a model that has been matured on the Morello platform for more than three years [38]. Supporting library compartmentalisation in CHERI-RISC-V is crucial in order to implement core benefits of CHERI on RISC-V. My analysis concluded that none of the existing primitives present on CHERI-RISC-V is suitable for supporting library compartmentalisation on CHERI-RISC-V. Furthermore, it has become obvious that the *Executive-Restricted* approach used on the Morello platform cannot be directly implemented on CHERI-RISC-V due to the use of banked registers between exception states, a mechanism natively supported on Armv8 but not on RISC-V. Moreover, Morello's *Executive-Restricted* approach cannot be easily used by multiple simultaneous compartmentalisation mechanisms. Thus, I proposed and evaluated the Thread ID (Capability) approach. Running the library-compartmentalisation model has proved that the architectural primitives added are sufficient on CHERI-RISC-V. While the current implementation of library compartmentalisation led to higher overheads than on the Morello platform, these can be explained by non-optimised software of library compartmentalisation on CHERI-RISC-V. My results show that Hypothesis H.5 is true.

**Hypothesis H.6**

CHERI allows the definition of hardware compartments defined in the architecture and can be implemented to have a small impact on PPA metrics.

Defining hardware compartments is important in order to protect attacks between microarchitectural resources that correspond to different software-defined protection domains. In Chapter 6, I present a prototype implementation demonstrating that it is possible to implement hardware compartments defined in the architecture. This work is improved in Chapter 7, which presents CID sealing. This allows the definition of hardware compartments in CHERI capabilities. Furthermore, the CID sealing approach presents a significant performance improvement compared to the Partition and Tag Identifier (PTID) approach. My work on hardware compartments is a prototype and requires future work. Therefore, my work does not conclusively prove that hardware compartments can be implemented with only a small impact on power and area metrics, but demonstrates that the overheads are likely to be small.

**Hypothesis H.7**

CHERI hardware compartments can offer robustness against transient-execution attacks by implementing confidentiality and integrity, thus mitigating transient-execution attacks.

CHERI compartmentalisation offers great architectural security guarantees. In order to have an overall secure system, CHERI compartments need to be protected against transient-execution attacks. In Chapter 6, I present an implementation of tagging and partitioning. This enforces that during compartment execution a compartment can only read and write its own state therefore guaranteeing confidentiality and integrity. In Chapter 7, I improve this by enforcing the same guarantees on compartment transitions.

## 8.3 Future work

In this section, I identify fields for future work. All of the work touched upon clearly exceeds the scope of my doctoral research.

### 8.3.1 Bringing contracts to a new level

While a large chunk of my doctoral research focused on architectural speculation contracts, many research questions have only been lightly touched or have not been explored at all.

#### Formal reasoning

The contracts shown in Chapters 3 and 4 are presented in prose. In order to formally reason about them, their behaviour would need to be formally defined. This requires a great number of development cycles in order to non-intrusively describe the desired behaviour. A first starting point can be to follow the rule-based formalisation methodology of Guarnieri et al. [43]. Instead of attempting to model an entire processor, I suggest to model parts of a processor, e.g., the ALU. Once this has been facilitated, properties over the abstract definition of the contracts can be established and proved for at least parts of the processor.

#### Building secure software

The properties that can be proven with formal verification methods can serve as useful building blocks to secure software. With architectural speculation contracts offering guarantees, software can pick which set of guarantees it wants for a certain code region. Software can be configured with different security guarantees by using architectural mechanisms like barrier or fence instructions on top of the guarantees already being present through speculation contracts. This allows for a run time evaluation and picking the appropriate configuration for real-world use cases.

### 8.3.2 Compartment ID sealing

The ideas presented for CID sealing in Chapter 7 require more research in order to determine whether they can become a CHERI standard feature. This necessitates a compartmentalisation model fully incorporating them, e.g., library compartmentalisation on CHERI-RISC-V.

#### Software models

The architectural representation of a CID has been shaped by the current software need for it. However, the compartmentalisation models are still in development on CHERI and there has not yet been a thorough evaluation of which models will prevail. The choice of model also plays an important role on which entity allocates CIDs and to what extent they are unique[3]. This and other questions are connected to CID revocation. When CIDs

[3]An example question would be: are CIDs only unique in an OS process or possibly even for the entire userspace.

are no longer unique, software needs to ensure that no capability leaks can occur both architecturally as well as microarchitecturally.

### Possible performance impact

Furthermore, it has to be evaluated whether CID sealed capabilities cannot only be considered a security feature, but also a performance feature. A research hypothesis is that CID sealing can improve the performance of domain switches. In a case where a compartment does not need to clear its register state because it is sealed by default, it can save many instructions to execute. Another open question is what the performance wins of an improved register clearing sequence are, as demonstrated in Section 5.4.

# Appendix A

# Indirect sealed entries

Previous work on the CHERI architecture [78] as well as the experimental part of the CHERI ISA reference manual [93] lay out the rudimentary idea of indirect sealed entry capabilities. In this document, I refer to these as *indirect sentries*. It describes a capability that points to one or more capabilities, which are then used as the entry capabilities. These capabilities are sealed and thus only accessible via dedicated instructions. There exist two variants of indirect sentries, which I have introduced in Section 5.1.3. On CHERI-RISC-V, the work is very early with no instructions having been proposed.

A straightforward design would be to create an instruction each for both indirect sentry approaches. For points-to-PCC indirect sentries, there would be an instruction that unseals a data capability, loads from this capability, and jumps to the loaded code capability. For points-to-pair indirect sentries, there would be an instruction that loads two capabilities, jumps to the code capability and unseals the data capability. Both of these instructions have the design issue of not being *RISC-like* instructions, e.g., having multiple written registers where one is the norm. This might prevent them from being easily incorporated into microarchitectures.

In the following text, I present very early research and reasoning about how to implement indirect sentries, which leads to future work in this field.

# A.1 Points-to-PCC

In this section, I am designing instructions for points-to-PCC indirect sentries. Furthermore, I explain the security guarantees they give.

## A.1.1 Instruction design

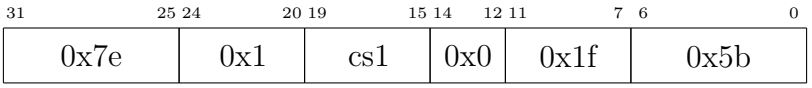
As explained above, designing a complex instruction for points-to-pair indirect sentries heavily impacts microarchitectural design and thus is not desirable. The points-to-PCC invoking instruction does three operations:

1. Unseal the data capability and load the code capability.
2. Install the code capability in the PCC register.
3. Store the data capability into the IDC register.

I am proposing to create one instruction for operation 1 and one instruction for operations 2 and 3: `CLIL` and `CJAURL`. This pair of instructions are linked. Only if both instructions finish successfully, a jump to an indirect sentry is completed.

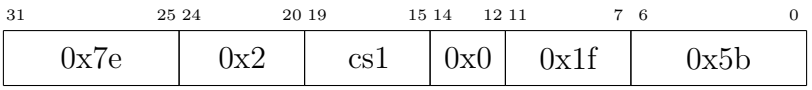
Furthermore, there needs to be an instruction that produces a valid indirect sentry. This instruction needs to be in style of the `CSealEntry` instruction presented in the CHERI-RISC-V ISA [93]. The specification of this instruction remains future work.

CLIL cs1




`CLIL` (CHERI load indirect sentry linked) loads the code capability from the indirect sentry in *cs1*, seals the code capability with the otype for points-to-PCC linked loads[1], and stores the code capability in the implicit destination register, which is register `c31`. The instruction creates a reservation for the indirect sentry stored in *cs1*. The instruction will fail on multiple conditions. One is if the capability loaded is sealed: this case would allow "re-sealing" via the `CLIL` instruction, which would lead to capability leaks.

CJAURL cs1




[1]This is a dedicated otype value expressing that this capability is sealed after invoking a `CLIL` instruction. This otype is not available to conventional sealed pairs of capabilities.

CJAURL (CHERI jump and and unseal register linked) unseals the indirect sentry stored in *cs1* into the IDC register and installs the code capability stored in the IDC register into the PCC register. This instruction evaluates to a NOP (No-Operation) if an existing reservation does not match *cs1* or if no reservation is present. However, if the `CJAURL` includes an invalid code capability, the instruction will raise an exception. The exact conditions under which the instruction evaluates to a NOP and under which it raises an exception are still to be specified.

### A.1.2 Security considerations

The original proposal of having one instruction for an indirect sentry jump guarantees atomicity and thus gives the security guarantees of either both the unsealing of the data capability and the jump to the code capability happens or none of the two happen. However, when splitting this in two instructions, one needs to reason about the security guarantees still being upheld.

While the `CLIL` instruction loads the code capability, code cannot use it without the `CJAURL` instruction. The capability is sealed with a dedicated otype and thus cannot be unsealed via a `CUnseal` instruction and neither can be used as a sentry [93]. The code capability of the indirect sentry now becomes readable. However, I do not consider that a security issue. An attacker does not gain any privileges to the compartment, but only learns where the code is allocated and which permissions it has. Furthermore, the same security assumption is used by the pair of capabilities used via the `CInvoke` instruction already present in the CHERI v9 ISA [93].

One attack possibility is to attempt to link a code capability and a data capability that were not linked in the original indirect sentry. This would allow attackers to gain privileges in various ways, e.g., leaking the data capability to the attackers code. This and other scenarios are disallowed by the reservation. The reservation ensures that the same indirect sentry is used for both instructions. Using a different register index as the indirect sentry in the `CJAURL` instruction will make the instruction evaluate to a NOP. Furthermore, writing to the register that holds the indirect sentry, will invalidate the reservation. Equally, writing to the IDC register will invalidate the reservation.

### A.1.3 Code considerations

The `CJAURL` instruction does not always succeed. This can happen if the instruction sequence does not follow the order of instructions laid out by the programmer, which can be caused by interrupts. Thus, the two instructions need to be re-run until they succeed, which can be compared to linked instructions for atomic memory operations. I laid out the code pattern for the pair of linked points-to-PCC indirect sentry instructions in Figure A.1.

```
end_of_function:
  clil ct5            // Load from indirect sentry
  ...                 // Useful work in between
  cjaurl ct5          // Attempt jump to code capability

// The below will not be executed if the jump was successful
repeat:
  clil ct5            // Load again
  cjaurl ct5          // Attempt to jump again
  beq x0, x0, repeat // Try until success
```

Figure A.1: Code pattern for doing domain crossings with linked points-to-PCC indirect sentry instructions.

The first pair of linked instructions is split by useful work to be done, e.g., re-saving register values from the stack. If the `CJAURL` succeeds, the jump to the indirect sentry has been successful and the code below will never be executed. If the jump evaluated to a NOP, the second pair of linked instructions attempts to perform the indirect sentry jump. If this is not successful, the branch instruction will jump two instructions back and try again until it finally succeeds.

## A.2 Points-to-pair

I have not yet specified instructions for points-to-pair indirect sentries. One potential research route could be a triplet of linked instructions: one instruction for loading the data capability, one instruction for loading the data capability, and one instruction for the jump.

# Bibliography


[1] Sam Ainsworth and Timothy M. Jones. "MuonTrap: Preventing cross-domain Spectre-like attacks by capturing speculative state". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, 2020, pp. 132–144. ISBN: 9781728146614. DOI: 10.1109/ISCA45697.2020.00022. URL: https://doi.org/10.1109/ISCA45697.2020.00022.

[2] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Wesley Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. *CHERIoT: Rethinking security for low-cost embedded systems*. Tech. rep. MSR-TR-2023-6. Microsoft, Feb. 2023. URL: https://www.microsoft.com/en-us/research/uploads/prod/2023/02/cheriot-63e11a4f1e629.pdf.

[3] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo Wesley, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. "CHERIoT: Complete memory safety for embedded devices". In: *proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture*. MICRO '23. Toronto, ON, Canada: Association for Computing Machinery, Oct. 2023, pp. 641–653. DOI: https://doi.org/10.1145/3613424.3614266.

[4] Amazon Web Services. *AWS graviton performance testing: Tips for independent software vendors*. Tech. rep. 2024, p. 34. URL: https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-graviton-performance-testing/aws-graviton-performance-testing.pdf#what-is-aws-graviton.

[5] AMD. *Speculation behavior in AMD micro-architectures*. Tech. rep. 5.14.19. May 2019. URL: https://www.amd.com/system/files/documents/security-whitepaper.pdf.

[6] Arm Limited. *Arm architecture reference manual supplement Morello for A-profile architecture*. Tech. rep. DDI0606, Version: A.k. Jan. 2022. URL: https://developer.arm.com/documentation/den0133/0100/Arm-Morello-implementation/Instruction-Set-Architecture.

[7] Arm Limited. *Arm Morello program*. https://www.arm.com/architecture/cpu/morello. May 2022.

[8] Arm Limited. *Arm Neoverse N1 CPU: Accelerating the transformation to a scalable cloud to edge infrastructure*. Feb. 2019.

[9] Arm Limited. *Arm Neoverse v1 platform: A revolution in high performance computing*. Apr. 2021.

[10] Arm Limited. *Arm v8.5-A/v9 CPU updates*. Tech. rep. 1.5. May 2021, p. 12. URL: https://developer.arm.com/documentation/102822/latest/.

[11] Arm Limited. *Cache Speculation Side-channels*. Tech. rep. 2.5. June 2020, p. 21. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[12] Arm Limited. *Cortex-A53*. https://developer.arm.com/processors/cortex-a53. Aug. 2024.

[13] Arm Limited. *Cortex-A72 software optimization guide*. Mar. 2015.

[14] Arm Limited. *Stack Pointer register*. https://developer.arm.com/documentation/dui0801/l/Overview-of-AArch64-state/Stack-Pointer-register?lang=en. 2024.

[15] Arm Limited. *Straight-line speculation*. Tech. rep. 1.0. June 2020, p. 6. URL: https://developer.%20arm.com/support/arm-security-updates/speculative-processorvulnerability/downloads/straight-line-speculation.

[16] Arm Limited. *Vulnerability of speculative processors to cache timing side-channel mechanism*. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. June 2022.

[17] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[18] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. "MI6: Secure enclaves in a speculative out-of-order processor". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 42–56. ISBN: 9781450369381. DOI: 10.1145/3352460.3358310. URL: https://doi.org/10.1145/3352460.3358310.

[19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium*. Ed. by William Enck

and Adrienne Porter Felt. USENIX Association, 2018, pp. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking transient execution through microarchitectural load value injection". In: *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2020, pp. 54–72. DOI: 10.1109/SP40000.2020.00089. URL: https://doi.org/10.1109/SP40000.2020.00089.

[21] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking data on meltdown-resistant CPUs". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 769–784. DOI: 10.1145/3319535.3363219. URL: https://doi.org/10.1145/3319535.3363219.

[22] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A systematic evaluation of transient execution attacks and defenses". In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 249–266. ISBN: 9781939133069.

[23] Chandler Carruth. *Speculative Load Hardening*. https://llvm.org/docs/SpeculativeLoadHardening.html. 2018.

[24] Mike Clark. "A new ×86 core architecture for the next generation of computing". In: *2016 IEEE Hot Chips 28 Symposium (HCS)*. IEEE Computer Society. 2016, pp. 1–19.

[25] Codasip GmbH. *CHERI security technology*. https://codasip.com/solutions/riscv-processor-safety-security/cheri/. Nov. 2023.

[26] The kernel development community. https://www.kernel.org/doc/html/latest/x86/pti.html. May 2022.

[27] Jonathan Corbet. *The current state of kernel page-table isolation*. https://lwn.net/Articles/741878/. Dec. 2017.

[28] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. "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*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 379–393. ISBN: 9781450362405. DOI: 10.1145/3297858.3304042. URL: https://doi.org/10.1145/3297858.3304042.

[29] Editors: Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume I*. Version 20240411. Unprivileged Architecture. RISC-V International. Apr. 2024.

[30] Editors: Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume II*. Version 20240411. Privileged Architecture. RISC-V International. Apr. 2024.

[31] Embedded Microprocessor Benchmark Consortium. *CoreMark*. https://www.eembc.org/coremark/. July 2022.

[32] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. "BranchScope: A new side-channel attack on directional branch predictor". In: *SIGPLAN Not.* 53.2 (Mar. 2018), pp. 693–707. ISSN: 0362-1340. DOI: 10.1145/3296957.3173204. URL: https://doi.org/10.1145/3296957.3173204.

[33] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. "Cornucopia: Temporal safety for CHERI heaps". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2020, pp. 1507–1524.

[34] Franz A. Fuchs. *Analysis of transient-execution attacks on the out-of-order CHERI-RISC-V microprocessor Toooba*. Tech. rep. KTH Royal Institute of Technology, 2021, p. 72.

[35] Franz A. Fuchs, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. "Developing a test suite for transient-execution attacks on RISC-V and CHERI-RISC-V". In: *Workshop on Computer Architecture Research with RISC-V*. June 2021.

[36] Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Alexandre Joannou, Jessica Clarke, John Baldwin, Brooks Davis, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. "Safe speculation for CHERI". In: *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2024.

[37] Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Marno van der Maas, Alexandre Joannou, Alexander Richardson, Jessica Clarke, Nathaniel Wesley Filardo, Brooks Davis, John Baldwin, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. "Architectural contracts for safe speculation". In: *2023 IEEE 41st International Conference on Computer Design (ICCD)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2023, pp. 578–586. DOI: 10.1109/ICCD58817.2023.00093. URL: https://doi.ieeecomputersociety.org/10.1109/ICCD58817.2023.00093.

[38] Dapeng Gao and Robert N. M. Watson. "Library-based compartmentalisation on CHERI". In: *Programming Languages for Architecture 2023*. Orlando, FL, USA, June 2023.

[39] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* 8 (Apr. 2018), pp. 1–27. DOI: 10.1007/s13389-016-0141-6.

[40] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. "IntroSpectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities". In: *48th Annual International Symposium on Computer Architecture (ISCA)*. June 2021, pp. 874–887.

[41] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. "The Arm Morello evaluation platform—Validating CHERI-based security in a high-performance system". In: *IEEE Micro* 43.3 (2023), pp. 50–57.

[42] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled detection of speculative information flows". In: *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society. 2020, pp. 1–19.

[43] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-software contracts for secure speculation". In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society. 2021.

[44] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alexander Richardson. "Clean application compartmentalization with SOAAP". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1016–1031. ISBN: 9781450338325. DOI: 10.1145/2810103.2813611. URL: https://doi.org/10.1145/2810103.2813611.

[45] Jan Horn. *speculative execution, variant 4: speculative store bypass*. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Feb. 2018.

[46] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual*. Tech. rep. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Apr. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[47] Intel Corporation. *Intel analysis of speculative execution side channels*. Tech. rep. 1.0. Jan. 2018.

[48] Intel Corporation. *Retpoline: A branch target injection mitigation*. Tech. rep. June 2018. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf.

[49] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. "Cyclone: A safe dialect of C". In: *2002 USENIX Annual Technical Conference (USENIX ATC 02)*. Monterey, CA: USENIX Association, June 2002.

[50] Alexandre Joannou, Peter Rugg, Jonathan Woodruff, Franz A. Fuchs, Marno van der Maas, Matthew Naylor, Michael Roe, Robert N. M. Watson, Peter G. Neumann, and Simon W. Moore. "Randomized testing of RISC-V CPUs using direct instruction injection". In: *IEEE Design & Test* 41.1 (Feb. 2024), pp. 40–49. DOI: 10.1109/MDAT.2023.3262741.

[51] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. "Efficient tagged memory". In: *2017 IEEE International Conference on Computer Design (ICCD)*. Boston, MA, USA, Nov. 2017, pp. 641–648. DOI: 10.1109/ICCD.2017.112.

[52] David Kaplan, Jeremy Powell, and Tom Woller. *AMD SEV-SNP: Strengthening VM isolation with integrity protection and more*. Tech. rep. Advanced Micro Devices Inc., Jan. 2020. URL: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[53] Paul A Karger. "Limiting the damage potential of discretionary Trojan horses". In: *1987 IEEE Symposium on Security and Privacy*. IEEE. 1987, pp. 32–32.

[54] Shivangi Kharoo. *RISC-V market trends And predictions till 2030*. https://www.electronicsb2b.com/eb-specials/risc-v-market-trends-predictions-till-2030/. June 2024.

[55] Douglas Kilpatrick. "Privman: A library for partitioning applications." In: *USENIX Annual Technical Conference, FREENIX Track*. 2003, pp. 273–284.

[56] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372. ISBN: 9781479943944.

[57] Paul Kocher. *Spectre mitigations in Microsoft's C/C++ compiler*. `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html`. Feb. 2018.

[58] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2019, pp. 1–19. DOI: `10.1109/SP.2019.00002`. URL: `https://doi.org/10.1109/SP.2019.00002`.

[59] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential power analysis". In: *Proceedings of the 19th Annual International Cryptology Conference*. Crypto '99. Santa Barbara, CA, USA: Springer Berlin, Aug. 1999, pp. 388–397.

[60] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. "Spectre returns! Speculation attacks using the return stack buffer". In: *12th USENIX Workshop on Offensive Technologies, WOOT 2018*. Ed. by Christian Rossow and Yves Younan. USENIX Association, 2018. URL: `https://www.usenix.org/conference/woot18/presentation/koruyeh`.

[61] Chester Lam. *Intel's Netburst: Failure is a foundation for success*. `https://chipsandcheese.com/2022/06/17/intels-netburst-failure-is-a-foundation-for-success/`. June 2022.

[62] David Lecomber. *Using Arm servers to reduce the time and cost of genomics*. `https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/aws-graviton3-reduces-time-and-cost-for-genomics`. Oct. 2022.

[63] Linaro Limited. *Morello platform open source software*. `https://www.morello-project.org`. Sept. 2020.

[64] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading kernel memory from user space". In: *27th USENIX Security Symposium*. Aug. 2018.

[65] Matthias Ludwig, Alexander Hepp, Michaela Brunner, and Johanna Baehr. "CRESS: Framework for vulnerability assessment of attack scenarios in hardware reverse engineering". In: *2021 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. Washington, DC, USA: IEEE, Nov. 2021, pp. 1–8. DOI: `10.1109/PAINE54418.2021.9707695`.

[66] Giorgi Maisuradze and Christian Rossow. "ret2spec: Speculative execution using return stack buffers". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, Jan. 2018, pp. 2109–2122. ISBN: 9781450356930. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761.

[67] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. "Going beyond the limits of SFI: Flexible and secure hardware-assisted in-process isolation with HFI". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 266–281. ISBN: 9781450399180. DOI: 10.1145/3582016.3582023. URL: https://doi.org/10.1145/3582016.3582023.

[68] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. "Revizor: Testing black-box CPUs against speculation contracts". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2022. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 226–239. ISBN: 9781450392051. DOI: 10.1145/3503222.3507729. URL: https://doi.org/10.1145/3503222.3507729.

[69] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. *Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches*. 2017. arXiv: 1702.00719 [cs.CR]. URL: https://arxiv.org/abs/1702.00719.

[70] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: The case of AES". In: CT-RSA'06. San Jose, CA, USA: Springer-Verlag, 2006, pp. 1–20. ISBN: 3540310339. DOI: 10.1007/11605805_1. URL: https://doi.org/10.1007/11605805_1.

[71] David A. Patterson and John L. Hennessy. *Computer organization and design RISC-V edition*. Morgan Kaufmann, May 2017. ISBN: 9780128122761.

[72] Filip Pizlo. *What Spectre and Meltdown mean for WebKit*. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/. Jan. 2018.

[73] Niels Provos, Markus Friedl, and Peter Honeyman. "Preventing privilege escalation." In: *USENIX Security Symposium*. 2003.

[74] Allison Randal. *Transient execution vulnerabilities in the security context of server hardware*. Tech. rep. UCAM-CL-TR-992. University of Cambridge, Computer Laboratory, Dec. 2023. DOI: 10.48456/tr-992. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-992.pdf.

[75] Teresa Reidt. *ARM vs. x86: Differences & similarities of both architectures*. https://emteria.com/blog/arm-vs-x86. July 2023.

[76] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site isolation: Process separation for web sites within the browser". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1661–1678. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/reis.

[77] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. "I see dead $\mu$ops: Leaking secrets via Intel/AMD micro-op caches". In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA '21. Virtual Event, Spain: IEEE Press, June 2021, pp. 361–374. ISBN: 9781450390866. DOI: 10.1109/ISCA52012.2021.00036. URL: https://doi.org/10.1109/ISCA52012.2021.00036.

[78] Alexander Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. Tech. rep. UCAM-CL-TR-949. University of Cambridge, Computer Laboratory, June 2020. DOI: 10.48456/tr-949. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf.

[79] Peter Rugg, Alexandre Joannou, Marno van der Maas, Jonathan Woodruff, Alexander Richardson, and Jessica Clarke. *TestRIG*. https://github.com/CTSRD-CHERI/TestRIG. Mar. 2024.

[80] Peter Rugg, Rishiyur S. Nikhil, Jessica Clarke, Niraj Sharma, Jonathan Woodruff, and Darius Rad. https://github.com/CTSRD-CHERI/Flute. 2024.

[81] Peter Rugg, Rishiyur S. Nikhil, Alexandre Joannou, Jessica Clarke, Niraj Sharma, and Jonathan Woodruff. https://github.com/CTSRD-CHERI/Piccolo. 2022.

[82] Peter Rugg, Jonathan Woodruff, Joannou Alexandre, and Simon W. Moore. "A Suite of Processors to Explore CHERI-RISC-V Microarchitecture". In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. Aug. 2024, pp. 351–360. DOI: 10.1109/DSD64264.2024.00054.

[83] Peter David Rugg. *Efficient spatial and temporal safety for microcontrollers and application-class processors*. Tech. rep. UCAM-CL-TR-984. University of Cambridge, Computer Laboratory, July 2023. DOI: 10.48456/tr-984. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-984.pdf.

[84] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue in-flight data load". In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2019, pp. 88–105. DOI: 10.1109/SP.2019.00087. URL: https://doi.org/10.1109/SP.2019.00087.

[85] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 753–768. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252.

[86] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. "Spectre declassified: Reading from the right place at the wrong time". In: *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, 2023, pp. 1753–1770. DOI: 10.1109/SP46215.2023.10179355.

[87] Standard Performance Evaluation Corporation. *SPEC CPU 2006*. https://www.spec.org/cpu2006/. Jan. 2018.

[88] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU register state using microarchitectural side-channels". In: *CoRR* abs/1806.07480 (2018). arXiv: 1806.07480. URL: http://arxiv.org/abs/1806.07480.

[89] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient software-based fault isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: Association for Computing Machinery, 1993, pp. 203–216. ISBN: 0897916328. DOI: 10.1145/168619.168635. URL: https://doi.org/10.1145/168619.168635.

[90] Robert N. M. Watson. *CHERITech2024 Talk 1.1 Welcome, introduction and demonstration*. https://www.youtube.com/watch?v=ayuA0jQAHrI. Apr. 2024.

[91] Robert N. M. Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W. Moore, Peter G. Neumann, Alexander Richardson, Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff. "CHERI: Hardware-enabled C/C++ memory protection at scale". In: *IEEE Security & Privacy* 22.04 (July 2024), pp. 50–61. ISSN: 1558-4046. DOI: 10.1109/MSEC.2024.3396701.

[92] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. *Early performance results from the prototype Morello microarchitecture*. Tech. rep. UCAM-CL-TR-986. University of Cambridge, Computer Laboratory, Sept. 2023. DOI: 10.48456/tr-986. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.pdf.

[93] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore

Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Tech. rep. UCAM-CL-TR-987. University of Cambridge, Computer Laboratory, Sept. 2023. DOI: 10.48456/tr-987. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf.

[94] Robert N. M. Watson, Jonathan Woodruff, Alexandre Joannou, Simon W. Moore, Peter Sewell, and Arm Limited. *DSbD CHERI and Morello Capability Essential IP (Version 1)*. Tech. rep. UCAM-CL-TR-953. University of Cambridge, Computer Laboratory, Dec. 2020. DOI: 10.48456/tr-953. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-953.pdf.

[95] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A hybrid capability-system architecture for scalable software compartmentalization". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. USA: IEEE Computer Society, 2015, pp. 20–37. ISBN: 9781467369497. DOI: 10.1109/SP.2015.9. URL: https://doi.org/10.1109/SP.2015.9.

[96] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, and Peter G. Neumann. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre attacks*. Tech. rep. UCAM-CL-TR-916. University of Cambridge, Computer Laboratory, Feb. 2018. DOI: 10.48456/tr-916. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf.

[97] Markus Wegmann, Andreas Traber, Pirmin Vogel, Sven Stucki, Rupert Swarbrick, and Philipp Wagner. https://github.com/CTSRD-CHERI/ibex. 2022.

[98] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. "NDA: Preventing speculative execution attacks at their source". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 572–586.

[99] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*. Tech. rep. 1.0. Aug. 2018, p. 7. URL: https://foreshadowattack.eu/foreshadow-NG.pdf.

[100] Johannes Wikner and Kaveh Razavi. "RETBLEED: Arbitrary speculative code execution with return instructions". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3825–3842.

[101] Konrad Witaszczyk. "Code-centered kernel compartmentalization in CheriBSD". In: *CHERITech'24*. Cambridge, UK, Apr. 2024.

[102] Claire Wolf. *RISC-V Formal verification framework*. `https://github.com/SymbioticEDA/riscv-formal`. 2018.

[103] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. "CHERI Concentrate: Practical compressed capabilities". In: *IEEE Transactions on Computers* 68.10 (2019), pp. 1455–1469.

[104] Jonathan Woodruff, Peter Rugg, Jessica Clarke, Franz A. Fuchs, Rishiyur S. Nikhil, and Marno van der Maas. `https://github.com/CTSRD-CHERI/Toooba`. 2024.

[105] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA*. IEEE Computer Society, 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201. URL: https://doi.org/10.1109/ISCA.2014.6853201.

[106] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel Wesley Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. "CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 545–557. ISBN: 9781450369381. DOI: 10.1145/3352460.3358288. URL: https://doi.org/10.1145/3352460.3358288.

[107] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. "InvisiSpec: Making speculative execution invisible in the cache hierarchy". In: *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 428–441.

[108] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 9781931971157.

[109] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. "Speculative Taint Tracking (STT) A comprehensive protection for speculatively accessed data". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA: Association for Computing Machinery, Oct. 2019, pp. 954–968.

[110] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. "Composable building blocks to open up processor design". In: *51st Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO)*. IEEE Computer Society, Oct. 2018, pp. 68–81. DOI: 10.1109/MICRO.2018.00015. URL: https://doi.org/10.1109/MICRO.2018.00015.

[111] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. "Ultimate SLH: Taking speculative load hardening to the next level". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7125–7142. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh.