Technical Report

UCAM-CL-TR-887
ISSN 1476-2986

Number 887

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Hardware support for compartmentalisation

Robert M. Norton

May 2016

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

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

This technical report is based on a dissertation submitted September 2015 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare Hall.

With thanks to Microsoft Research Limited who provided the primary funding for this work via their PhD scholarship program under contract MRL-2011-031.

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and FA8750-11-C-0249 ("MRC2") as part of the DARPA CRASH and DARPA MRC research programs. The views, opinions, and/or findings contained in this report are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

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

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

ISSN 1476-2986

Abstract

Compartmentalisation is a technique to reduce the impact of security bugs by enforcing the 'principle of least privilege' within applications. Splitting programs into separate components that each operate with minimal access to resources means that a vulnerability in one part is prevented from affecting the whole. However, the performance costs and development effort of doing this have so far prevented widespread deployment of compartmentalisation, despite the increasingly apparent need for better computer security. A major obstacle to deployment is that existing compartmentalisation techniques rely either on virtual memory hardware or pure software to enforce separation, both of which have severe performance implications and complicate the task of developing compartmentalised applications.

CHERI (Capability Hardware Enhanced RISC Instructions) is a research project which aims to improve computer security by allowing software to precisely express its memory access requirements using hardware support for bounded, unforgeable pointers known as *capabilities*. One consequence of this approach is that a single virtual address space can be divided into many independent compartments, with very efficient transitions and data sharing between them.

This dissertation analyses the compartmentalisation features of the CHERI Instruction Set Architecture (ISA). It includes: a summary of the CHERI ISA, particularly its compartmentalisation features; a description of a multithreaded CHERI CPU which runs the FreeBSD Operating System; the results of benchmarks that compare the characteristics of hardware supported compartmentalisation with traditional techniques; and an evaluation of proposed optimisations to the CHERI ISA to further improve domain crossing efficiency.

I find that the CHERI ISA provides extremely efficient, practical support for compartmentalisation and that there are opportunities for further optimisation if even lower overhead is required in the future.

Acknowledgements

I would like to thank my supervisor, Simon Moore, for his advice and kindness throughout my PhD, as well as all other members of the CHERI project which have made this work possible through their engineering effort, advice and companionship. In particular, I would like to thank Robert Watson for his inspiration and leadership of the project, as well as his considerable engineering effort advancing FreeBSD on CHERI and, at times, helping to diagnose bugs in the CHERI2 processor. Brooks Davis, Stacey Son and Munraj Vadera were among those whose software engineering work was invaluable to the project as a whole, with Brooks deserving a special mention for helping to get symmetric multiprocessing support for CHERI2 to work with FreeBSD. David Chisnall acted as the compiler lynchpin, providing the necessary infrastructure for software development on the platform and always had astute observations to offer too. Mike Roe and Theo Markettos's testing and specification work was crucial to keeping things working together, especially as the project gained momentum. Jonathan Woodruff and the original CHERI processor provided implementation ideas and a critical ear, whilst Nirav Dave's initial implementation of CHERI2 provided a solid base on which to build, as well as tips and tricks for programming in Bluespec. Alan Mujumdar, Colin Rothwell, Alex Horsman, and Alexandre Joannou were always ready to hear about my latest difficulty or discuss new possibilities as we worked together on shared infrastructure. The importance of Peter Neumann's historical perspective should not be underestimated, and Andrew Moore's generous coffee provision was indispensable. The work would not have been possible without the support of sponsors including Microsoft Research, who funded my PhD scholarship, and the Defense Advanced Research Programs Agency, whose support of the CTSRD and MRC2 projects was fundamental to the development of CHERI. Finally, I owe a great debt to my wonderful wife, Jenny, for supporting me and enduring lonely times with our lovely daughter, Freya, whenever deadlines loomed.

Contents

CHAPTER 1

Introduction

1.1 The need for computer security

Computer systems have become integral to peoples' daily lives and a critical part of modern infrastructure, so computer security is more crucial than ever. Numerous high profile attacks and vulnerabilities demonstrate that current techniques are inadequate to protect against the threats of the modern digital world. For example the Stuxnet worm targeted and compromised Iranian nuclear facilities [1], and the Heartbleed bug exposed millions of systems to remote exploitation [2]. Even the fear of cyber attack is sufficient to intimidate large corporations, as in the case of Sony Pictures' decision to edit scenes and cancel the New York premiere of its production 'The Interview' in response to hackers' threats [3]. Meanwhile millions of personal and business computers are compromised daily, resulting in privacy concerns and financial losses [4].

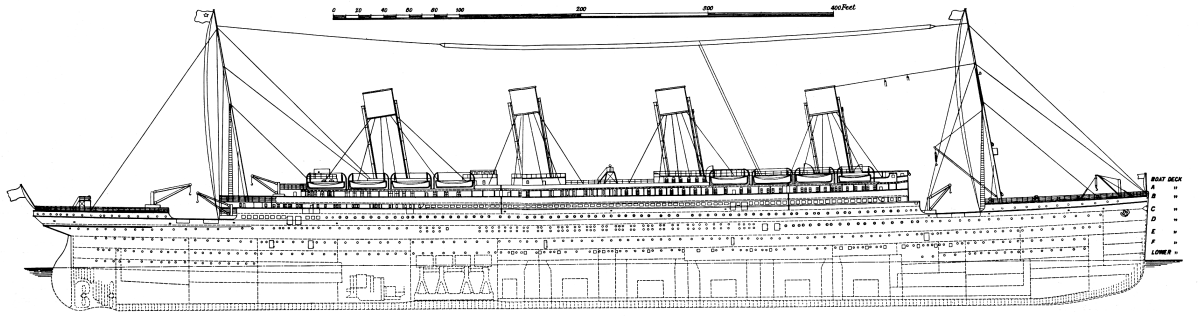
1.2 Part of the solution: compartmentalisation

Many tools and strategies exist for reducing the incidence of vulnerabilities in software, but, as evidenced by common vulnerability exposure (CVE) statistics, exploitable bugs inevitably remain [6]. Szekeres et al. [7] make it clear in their systematisation of knowledge paper 'Eternal War in Memory' that many real world exploits are caused by the lack of memory safety in low-level languages such as C and C++. They also show that attack prevention techniques, such as stack cookies and address space randomisation, are never 100% effective. Therefore we need mitigation techniques to restrict the consequences of any compromise.

Compartmentalisation is one approach: applications are divided into subcomponents (often called a sandbox, compartment or security domain) following the *principle of least privilege*, where each part has access to the minimum set of resources required to accomplish its task. If one part of an application is compromised then only a limited part of the system is affected. This is often sufficient to prevent an attacker from exploiting a vulnerability or escalating a minor compromise into a more wide reaching one. Karger [8] observed that it can even be used to defeat Trojan horse attacks, where malicious software is introduced into the supply chain.

A useful analogy is that of large ocean liners, such as the RMS Titanic (Figure 1.1), which for over a hundred years have been built with multiple internal compartments to

Figure 1.1: Large ocean vessels use compartmentalisation to reduce the risk that a hull breach or fire will destroy the entire vessel, as shown in this side plan of the RMS Titanic from the early 20th century [5]. 16 watertight compartments were contained in the lower deck of the vessel, including some dedicated to the boilers and machinery.



contain flooding and fire. Although this might seem like an ill-chosen example given the tragic sinking of that ship on her maiden voyage on 31st May 1911, it actually serves to emphasise the importance of well-designed compartmentalisation.

The Titanic struck an iceberg which breached five of her sixteen compartments due to the failure of rivets along the starboard side in the bow. She was designed to survive up to four flooded compartments but more than this caused her to tilt, allowing water to flood over the top of the compartment walls and leading to her demise. Had the Titanic been designed with more compartments, for example with longitudinal divisions as well as lateral ones, or had the compartments been fully watertight she may not have sunk and over 1500 lives may have been saved.

Nevertheless compartmentalisation certainly slowed her sinking, allowing some passengers to escape, and the principle is still the mainstay of safety practice in marine engineering, such that the configuration and integrity of the compartments is one of the primary concerns when designing and maintaining ships [9].

Applying these lessons in the field of computers means dividing programs into many logical compartments and paying careful attention to the divisions between them. Achieving this requires the support of hardware to provide efficient isolation and communication between the parts.

1.2.1 The current state of the world

The value of compartmentalisation and least privilege in computer systems have been recognised since early days. The Honeywell 6180 computer used by Multics in late 60s supported a hierarchy of eight security 'rings' in hardware [10, 11]. This allowed the core part of the operating system to be isolated from less privileged parts such as drivers and applications, but left little scope for horizontal separation of parts within a ring. However, as Maurice Wilkes observed, just two rings are sufficient to isolate applications from each other when used in conjunction with memory protection hardware (e.g. segments or virtual memory):

> However, it eventually became clear that the hierarchical protection that rings provided did not closely match the requirements of the system programmer and gave little or no improvement on the simple system of having two

modes only. Rings of protection lent themselves to efficient implementation in hardware, but there was little else to be said for them [12].

This two- or three-level model has persisted to this day in the form of user, kernel and hypervisor modes of modern CPUs.

With the eventual consensus on paged virtual memory systems, which combine address space virtualisation with access control, application processes became the *de facto* unit of protection. This has been the situation for decades as these mechanisms are considered 'good enough' and there is a considerable body of research, experience and infrastructure built around them. OpenSSH was the first use of process based compartmentalisation in wide deployment [13] and now major web browsers, including Microsoft's Internet Explorer, Apple's Safari and Google's Chrome [14] all use processes to sandbox (isolate) high risk operations such as web page rendering and third-party plugins. However, as explored later in this dissertation, this is a relatively crude form of compartmentalisation that has major disadvantages in terms of performance and usability.

Software Fault Isolation (SFI) [15] as used by Google Native Client (NaCl) [16] and managed virtual machines (e.g. Microsoft's Common Language Runtime) are software-only techniques that attempt to work around the limitations of commodity hardware. Unfortunately these come with performance costs or limitations which make them unsuitable for use in many security critical contexts: OS kernels, web browsers and language runtimes themselves are routinely written in unmanaged languages, notably C and C++.

1.2.2 Research directions – where can we go from here?

To address this need for efficient compartmentalisation researchers have proposed novel processor architectures including Mondrian [17, 18], M-Machine [19], Iso-X [20], CODOMS [21] and CHERI [22–24]. These introduce new memory protection features that have support for fine-grained security domains. The suitability of these solutions for compartmentalisation depends greatly on the performance of security domain switching and, crucially, on the level of backwards compatibility for existing code bases. In this dissertation I consider each of these architectures, particularly focussing on CHERI and its practical support for compartmentalisation in hardware.

1.3 Contributions

This dissertations presents a number of contributions which I made to advance hardware support for compartmentalisation through the CHERI project:

- I conducted a survey of historical, currently used and proposed techniques for compartmentalisation (Chapter 2)
- I designed a multithreaded processor, CHERI2, implementing the CHERI ISA [25] and running on an FPGA. This processor supports the entire CHERI software stack, including CheriBSD, a port of the FreeBSD operating system, and supports cycle-accurate instruction tracing and user-space accessible counters for non-disruptive performance analysis at architectural and microarchitectural levels (Chapter 3).
- I designed a domain crossing micro-benchmark, `cheri_bench`, to compare the performance of process-based and CHERI supported compartmentalisation. (Section 4.1)

- I performed detailed, root-cause analysis of domain crossing performance using `cheri_bench` and CHERI2, looking at such features as the effect of the CHERI register set size on cache footprint during system calls (Section 4.3), the effect of TLB usage on the scalability of data transfer during domain crossing (Section 4.5), and the effect of hardware multithreading (Section 4.5).

- I designed, implemented and evaluated simple, practical hardware support to increase the efficiency of security domain crossing on the CHERI ISA through a new register clearing instruction (Section 4.6).

- I analysed the performance of `cheri_tcpdump`, an example of a real application adapted to use CHERI compartmentalisation, and demonstrated the scalability and practicality of the CHERI compartmentalisation features (Section 5.2).

- I contributed more generally to the engineering efforts of the project, including ISA-level testing, adapting the OS to support the hardware optimisations and implementing a tool to analyse instruction traces.

1.4 Publications

During the research period for this dissertation I contributed to two peer-reviewed papers with other members of the CHERI project.

- Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, **Robert Norton**, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st annual international symposium on computer architecuture*, pages 457–468. IEEE Press, 2014

- Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, J. Anderson, D. Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Stephen J. Murdoch, **Robert Norton**, Mike Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37, May 2015

Our ISCA 2014 paper [22] introduced the CHERI architecture and its support for fine-grained memory protection in the form of bounded, unforgeable pointers (capabilities), but did not focus on compartmentalisation features. It established that the runtime overheads of capabilities are acceptable and that efficient implementation in hardware is feasible.

Our IEEE S&P 2015 [23] paper elaborated on the compartmentalisation features of CHERI. Some of the results presented in Chapters 4 and 5 of this dissertation are included in this paper, but I present extended results, with more domain crossing mechanisms, more detailed analysis using hardware performance counters and new results from multithreaded hardware.

I also contributed to the following other papers and technical reports relating to the CHERI project:

- Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, **Robert Norton**, and Stacey Son. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, September 2015

- Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, **Robert Norton**, and Michael Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, April 2015

- Brooks Davis, **Robert Norton**, Jonathan Woodruff, and Robert NM Watson. How FreeBSD Boots: a soft-core MIPS perspective. In *Proceedings of AsiaBSDCon 2014*, March 2014

1.5 Dissertation overview

This dissertation is constructed as follows:

- Chapter 2 presents background material and previous work on compartmentalisation for security.

- Chapter 3 contains a description of the CHERI instruction set architecture and its support for compartmentalisation, as well as some details about the implementation of the CHERI2 processor and how compartmentalisation is supported by the llvm/clang compiler and FreeBSD operating system.

- Chapters 4 and 5 evaluate the implementation of compartmentalisation on CHERI at micro and macro levels respectively.

- Chapter 6 concludes the work and describes future research opportunities.

CHAPTER 2

Compartmentalisation for security

Compartmentalisation is the technique of separating computer applications into distinct parts to enhance security and reliability. This ensures that a failure or security compromise of one piece can be contained such that the system as a whole may continue. In this chapter I examine different approaches to compartmentalisation in hardware and software to understand how they meet the requirements for compartmentalisation and where there is room for improvement.

2.1 Compartmentalisation requirements

There are many methods of compartmentalisation, but they all have the following basic requirements:

Isolation This usually refers to memory protection and encompasses two considerations: *confidentiality* and *integrity*. These refer to whether a compartment may read and write another's data respectively. Depending on the trust relationship these properties may be needed in only one direction or in both. In situations where there is asymmetric distrust it is common to refer to the less trusted component as running in a *sandbox* (by analogy to a safe play area where no harm can be done). This dissertation deals mainly with mutual distrust between compartments or *security domains* which has stricter requirements, although the results are applicable to sandboxing too.

Resource arbitration On systems which multiplex hardware between multiple domains one must not be allowed to monopolise a resource preventing others from making progress. For example there needs to be a way to ensure that each domain gets a fair share of the CPU even in the presence of a misbehaving process. A corollary of this is that it must be possible for the application to detect and recover from the failure or misbehaviour of a compartment.

Access control In addition to arbitration there should be a means to control which resources a security domain can access. Policies vary but will commonly cover system services such as the file system and networking. The idea is to restrict a compartment to the minimum set of resources it needs to complete its task in accordance with the *principle of least privilege*.

Inter-domain communication Compartments need a way to communicate with each other so that, working together, they have all the resources they require. Where multiple security domains are time-multiplexed on the same hardware (e.g. with process based isolation on a single core) permitting controlled communication whilst maintaining isolation requires a *domain crossing* mechanism that allows transitioning from one security domain into another without granting all the permissions of one domain to the other. This typically involves a trusted component (e.g. an OS kernel) which is exempt from the usual isolation mechanisms so that it can save the state of one domain and restore another, and a controlled way of invoking this component, such as a system call. System calls are an example of a *call gate*, where *privilege escalation* occurs at the same time as execution passes to a known, trusted piece of code which is responsible for securely performing privileged actions (such as modifying virtual address mappings).

An example of communication in a compartmentalised system would be the privilege separated `sshd` [13], where inter-process communication (IPC) occurs over a UNIX socket between a privileged *monitor* process and a less privileged or *slave* process acting on behalf of the remote user. Each interaction requires marshalling data to a custom protocol and sending it over the socket, resulting in a system call into the OS to effect the domain transition.

In multicore systems several security domains may be executing simultaneously and communication between them can occur directly through shared memory, removing the need for a domain transitions to pass through a trusted authority. For example, the Chromium browser [14] uses shared memory to pass rendered webpages between the sandboxed renderer and main browser process. However, such communication between mutually distrustful parties requires special considerations to maintain security: all incoming messages must be validated and care must be taken to ensure that the sender cannot modify the data during or after validation and, as in all concurrent systems, careful synchronisation is required to ensure thread-safe, race-free behaviour.

2.2 Desirable compartmentalisation features

In addition to the necessary features detailed above compartmentalisation systems will be more widely adoptable if they provide the following:

Ease of use Restructuring applications for compartmentalisation can impose a significant burden on the programmer. In particular if the application needs to be split into multiple communicating processes this introduces all the well-documented problems associated with parallel programming, with the added complication that security invariants must be maintained. Excessive difficulty will deter adopters or reduce the level of compartmentalisation achieved. A good system will minimise the amount of restructuring required and provide tools and libraries to assist with refactoring and communication.

Incremental adoption Given the need to restructure applications it should ideally be possible to compartmentalise only high risk parts of an application initially and gain some benefit immediately. The level of compartmentalisation can then be increased

over time. It should not be necessary to completely rewrite large codebases in order to take advantage of the system.

Fine granularity Technical constraints can place a limit on the minimum amount of functionality which can be placed in a compartment. This can lead to undesirable sharing between logically separate security domains. For example in systems where memory is protected at page granularity (e.g. 4 kiB) it is not possible to protect small objects without suffering unacceptable memory and performance overheads due to alignment requirements.

Performance Compartmentalisation almost always introduces some performance cost. As Skeres et al. have observed [7] programmers will not adopt methods with excessive overhead or it may limit the degree of compartmentalisation achievable. In addition to the overhead incurred by the isolation mechanism itself, the costs of compartment management, domain switching and communication are all important factors. Any mechanism which requires frequent kernel intervention to manage trusted data structures is unlikely to perform well due to system call overhead, therefore user-space delegation of some responsibility is a highly desirable feature.

Scalability The desire for fine granularity leads to an increase in the number of compartments, which implies more switching and communication between domains. It is therefore crucial that performance scales well, potentially making use of the modern trend towards multicore hardware.

Small TCB The Trusted Computing Base (TCB) of a system is all of the hardware and software which must be correct for the desired security properties to be maintained. Given that most computer software contains bugs, a system with a large, complex TCB is more likely to contain flaws resulting in vulnerabilities than one with a minimal TCB. A vulnerability in the TCB is typically more serious than a flaw in any one application as it compromises the whole system and not just a single application. Furthermore correcting the TCB bug will likely be more disruptive as it often requires upgrading, rebooting or even replacing (in the case of hardware TCB components) the system. Therefore a small, simple, easily verifiable TCB is highly desirable.

Flexibility Many compartmentalisation techniques are optimised for a particular use case. For example, it is common to assume a single controlling process instantiating a number of distrusted sandboxes. Whilst it is important to support this scenario a general purpose compartmentalisation system should also allow more complex, non-hierarchical relationships.

2.3 Existing compartmentalisation techniques

2.3.1 Virtual memory

The most widespread compartmentalisation systems in use today make use of the supervisor rings and virtual memory hardware available on contemporary CPUs. Each OS process constitutes a unit of isolation contained within its own virtual address space, with access to hardware and system resources mediated by the kernel.

Applications can be compartmentalised manually using standard discretionary access controls (e.g. users, groups and file permissions), and OS primitives such as pipes for communication, but this requires significant programmer effort and is error prone. Many security extensions have been developed to address this issue by simplifying the process of creating and managing sandboxes, some of which are described in the following sections.

2.3.2 OS Virtualisation

OS virtualisation is the processes of running multiple independent operating systems instances on the same hardware. It has grown in prominence recently as a low cost means of consolidating many previously separate servers onto increasingly powerful hardware, but can be used for compartmentalisation by creating pared-down OS instances dedicated to running a single application, reducing the chance that other applications running on the same hardware will be affected by any compromise without requiring the expense of multiple physical machines. Originally developed by IBM in the 1960s as a way to partition the resources of large mainframe computers, virtualised systems use a control program or *hypervisor* to supervise execution of multiple guest operating systems [28].

Then, in 1979, UNIX version 7 introduced the `chroot` system call allowing a process to change its view of the file system so that it appears to be rooted in a subdirectory, potentially limiting it to a minimal set of required files. Although this was originally intended to aid development and testing, not for security, by 1992 Bill Cheswick described its use to make a 'jail' to track the actions of a computer hacker [29]. Pure `chroot` only limits file system access and as such it is not a complete isolation mechanism on its own. Further developments such as Solaris 'containers' [30], Linux Containers (LXC), OpenVZ and FreeBSD 'jails' [31] extend the concept to include networking and other system functions necessary for full compartmentalisation, effectively virtualising the system at the OS/kernel level. This form of virtualisation can run on commodity hardware and has moderate overhead, but requires pervasive support throughout the OS kernel, resulting in a large attack surface.

Recently para-virtualisation as used in Xen [32] and hardware-assisted virtualisation such as Intel VT (Virtualisation Technology), AMD-V have returned to running an entire OS on a virtual machine under the control of a hypervisor. This has the advantage of a smaller and simpler TCB but has higher overhead than kernel-level virtualisation.

With OS virtualisation the compartments themselves are extremely heavyweight, requiring an entire system image, although copy-on-write or other file-system tricks such as overlays can mitigate this. As such, OS virtualisation can only be used for compartmentalisation at a very coarse granularity.

2.3.3 Sandbox Modes

A lighter-weight alternative to full OS virtualisation is a special sandbox mode where a process can voluntarily give up all but a very narrow set of rights. For example, on Linux, a process in `seccomp` (secure computing) mode can only invoke the `exit`, `sigreturn`, `read` and `write`, and only on file handles it already had before entering `seccomp` mode. Applications can be sandboxed using this by dividing them into multiple processes, usually with a single main process with full privileges and subordinate sandbox processes to do 'risky' computation in `seccomp` mode. Communication is via pipes or files opened before forking / entering `seccomp` mode.

This model has much smaller impact on the kernel than OS-kernel level virtualisation and a drastically reduced attack surface, but requires significant modification to the design of the application. Unlike virtualisation, it is only suitable for sandboxing functions which can be modified to work in the restricted environment, so may not be usable for 3rd-party plugins supplied without source code, or for things other than pure computation, such as network access.

`seccomp-bpf` extends `seccomp` with programmable in-kernel syscall filtering using Berkeley Packet Filters (BPF), a simple virtual machine instruction set originally designed for filtering network packets. This allows sandbox designers to define rules to determine allowable kernel invocations instead of the fixed set of four permitted syscalls, considerably increasing flexibility. `OpenSSH` [13] and `Chromium` [14] use this for sandboxing on Linux.

2.3.4 Capsicum

Capsicum [33] is a sandboxing framework present in FreeBSD 9 and later which focuses on simplifying the creation of sandboxed applications (or sandboxing existing applications) on contemporary OSes. It attempts to ease the task of programming sandboxed applications by introducing a restricted sandbox mode and a new primitive, *capabilities*, which extend standard UNIX file descriptors to provide fine-grained privilege delegation. Processes which enter capability mode are denied direct access to global namespaces such as the file system and process IDs. Instead they may only access resources through the capabilities they possess. Capabilities can be inherited from the parent process or passed between sandboxes via inter-process communication channels such as UNIX domain sockets. Capsicum also features a user space library, `libcapscium`, which aids developing and running sandboxed code by providing support for loading sandboxes and a run-time environment which emulates much of the traditional UNIX API inside the sandbox, potentially using remote procedure calls (RPC) to proxy some operations (such as DNS lookups) via a more privileged process.

Watson et al. evaluate the framework using microbenchmarks and by porting a number of applications including gzip, tcpdump and the chromium browser. They found that the overhead of the extra capability checks in system calls was minimal, but that the cost of creating sandboxes by using `fork` and `execve` was significant, a problem in common with other sandbox modes relying on process isolation. Porting applications required relatively few source code changes, especially in applications already written to support sandboxing such as chromium.

2.3.5 Mandatory Access Control Frameworks

The term mandatory access control (MAC) was originally used to describe non-discretionary access controls such as Bell LaPadula [34] used in multilevel secure (MLS) systems, where discretionary access control is augmented with a system-wide policy which restricts what users can do with data they can access. Such systems are principally used in military contexts, but the concept of a separate security policy has recently been applied more widely in MAC frameworks for commodity OSes such as AppArmor [35] and SELinux [36]. Like `seccomp` BPF filters, these frameworks allow application sandboxing by defining a set of rules, however the 'policy' is specified declaratively by referring to higher level concepts such as users, binaries and files (either via paths names or file system supported 'labels').

Policies can be defined and distributed separately from the program, allowing sandboxing of arbitrary, unmodified applications much like OS/kernel virtualisation.

Unfortunately these frameworks are complex, with a large attack surface, and determining the correct policy for a particular application is an expert task. It is not uncommon to see applications fail due to incorrect security policies, and sadly the most frequent 'fix' is to disable the security framework altogether.

2.3.6 Microkernels

The original development of the Mach [37] kernel was motivated principally by the need to support multicore architectures, but over time it developed into a *microkernel* system consisting of a minimal kernel supporting a system of processes communicating via interprocess communication (IPC) in order to perform all the work of a UNIX operating system. This effectively produced a compartmentalised system with a heavy dependence on efficient IPC for domain crossing, but Mach's performance was soon found to be a limiting factor, largely due to increased cache usage [38, 39].

Later microkernels such as EROS [40] and L4 [41] aimed to address this with highly optimised methods of crossing between domains, principally by dropping portability and optimising for particular hardware platforms. Liedtke achieved impressive results [42] with latencies as low as 45 cycles for 'simple IPC' on a DEC Alpha, rising to 86 and 121 cycles on MIPS R4600 and Intel Pentium processors respectively in the best case, with further 'indirect costs' incurred due to cache usage bringing the typical IPC to around 200 cycles. However, this 'simple IPC' is limited to data which can be transferred in registers: transferring larger data would require expensive TLB manipulation, with the cost of a page fault being given at 600 or more cycles per page.

Shapiro [43] measured 4.9 microseconds (over 500 cycles) on a Pentium processor for a minimal EROS IPC round-trip with no data transfer and this overhead quickly escalates (relative to a within address-space `bcopy`) for larger data transfers due to data copying, cache and TLB misses. Shapiro argues that the high fixed cost is acceptable providing appropriate domain-crossing boundaries are chosen to amortise them:

> Appropriate decomposition points are those places where a significant data payload must be copied across a functional boundary within the application, or where the routine invoked is long-running relative to the IPC cost (e.g. sorting).

However this fails to address the fact that copying large amounts of data (even using TLB re-mapping tricks) via shared memory will lead to significant overhead because each shared page requires two entries in the TLB (one in the sender and one in the receiver). The effect of this is even worse on modern multicore architectures where TLB modification of shared pages requires a TLB-shootdown operation, sending an inter-processor interrupt (IPI) to all CPUs to flush invalid entries from the TLB. It also places a burden on the programmer to identify such appropriate decomposition points which may not even exist in any given application!

2.3.7 Limitations of virtual memory based techniques

As each of the preceding virtual memory based techniques relies on address translation to provide isolation they share some fundamental limitations:

Page Granularity CPUs support pages of some fixed size, typically four kilobytes or more. This limits compartmentalisation to be rather coarse, at the level of whole programs or libraries rather than individual objects. In fact, as processors adopt larger page sizes of 4 MiB or even 1 GiB to improve TLB efficiency this problem will become worse.

TLB Performance A hardware Translation Lookaside Buffer (TLB) is used to cache the address mappings for each domain. On some architectures this must be flushed on every context switch resulting in many TLB misses each of which may take tens to hundreds of cycles to fill from memory using a hardware table walker or software trap handler. This makes domain switching a costly operation, even more so on ubiquitous multicore systems where it is necessary to inform the other cores using an inter-processor interrupt (a so called TLB shootdown).

More modern TLBs tag each entry with an address space identifier which allows entries from multiple processes to co-exist in the TLB. This can reduce misses when returning to a previously executed process after a switch but many misses will still occur, especially since TLB capacity is severely limited by the expense of implementing the content addressable structure in hardware. As the number of active processes increases and the time between transitions decreases the pressure on the TLB causes serious performance degradation as noted by McCurdy, Cox and Vetter [44].

System calls Control of the TLB must be a privileged operation, which means the kernel is involved in all compartment manipulation and domain transitions. This mandates the use of system calls which come with many associated costs: e.g. pipeline flushes, save and restore of register context and TLB, cache and branch predictor footprint .

To work around these limitations of TLB-based compartmentalisation, programmers have devised software-only techniques to enforce isolation and improve security.

2.3.8 Safe programming languages

Some languages embed compartmentalisation in the language model. For example in the object oriented languages Java and C# each class, method or field can specify its accessibility by other classes and packages. This makes each class into a security domain with method calls functioning as domain transitions. Java and C# also have security frameworks which can restrict access to system resources according to administrator provided policy. These features are implemented using software techniques such as compilers which insert dynamic checks or virtual machines (sometimes supporting just in time compilation) however, such managed runtime environments incur performance costs including garbage collection, permissions and bounds checks which make them unsuitable for some applications. As a result much code is written wholly or partly in low-level languages like C which compile directly to native machine instructions. Such native code cannot be the contained by the language runtime and can therefore subvert the security model, meaning it can only be used if trusted or contained using a technique such as software fault isolation (see Section 2.3.9).

The TCB of a safe programming environment is typically large, sometimes including the compiler or the entire language runtime comprising the virtual machine, code verifier,

libraries and security framework implementation. Historical vulnerabilities in Java and the Microsoft .NET framework have shown that such a large, complex TCB inevitably contains many opportunities for security critical bugs: a search for "Java SE" on the National Vulnerabilities Database website of the National Institute of Standards and Technology found 394 vulnerabilities posted in the past three years (as of September 2015), many of them with severity listed as 'high' and affecting the sandboxing features. Such bugs have seen large scale exploitation in recent years, for example the Flashback Trojan infected over 650,000 OS X computers by exploiting an unpatched vulnerability in the Java runtime [45]. Furthermore large bodies of existing code written in unsafe languages would need to be rewritten to take advantage of language safety, a barrier which has proved prohibitive to the universal adoption of safe languages despite high-profile attempts such as JavaOS [46, 47] or Microsoft Research's Singularity OS [48].

2.3.9 Software fault isolation

Wahbe [15] introduced a software-only mechanism for isolating native code within a single address space. This relies on static code analysis and inserted address checks to enforce separation of *fault domains* (sandboxes), working around the lack of hardware support for within address space isolation on MIPS and Alpha hardware. The proposed software fault isolation (SFI) technique uses the upper bits of addresses to divide the address space into segments and inserted runtime checks before each memory or control flow instruction to ensure that sandboxes only write data or jump to addresses within and their own segments. The inserted checks use dedicated registers to maintain the invariants even if a jump occurs directly to the memory access or branch instruction, bypassing the preceding check. Domain transitions are achieved by jumping via trusted trampolines, that can contain code which does not adhere to the normal SFI restrictions in order to achieve their work.

This version of SFI has a number of limitations, including that only memory writes are checked as reads are more common and not considered security critical. This facilitates read-only sharing of data between domains but means that there is no confidentiality between them, only integrity. Read-write data sharing between domains can be achieved by mapping physical memory at multiple locations in the address space so that it can appear in segments belonging to different domains, but this requires duplicate TLB entries and limits granularity. Another limitation is that only one code and one data segment are accessible in each domain.

Whabe's original SFI inspired a wide variety of successors including works focusing on x86 [16, 49–52], driver isolation [53–55] or native code executing in a Java virtual machine [56].

Implementations of SFI on commodity x86 are complicated by the fact that instructions are variable in size and can be unaligned, unlike the fixed 32-bit RISC instructions of the MIPS and Alpha machines used by Whabe. This makes static verification difficult as the verifier must ensure that even branches into the middle of an instruction do not violate the safety requirements. This can be worked around by replacing each indirect control flow operation with a lookup in a hash table of valid function entries as in MiSFIT [49], artificially aligning instructions with nops and masking the lower bits of indirect jumps as in PittSFIeld [51] or using unique labels embedded in the code as in control flow integrity [52, 57]. However each of these comes with additional runtime overhead and

requires a specialised toolchain to generate conforming code.

XFI [52] is an example of pure software fault isolation taken to an extreme. It uses a verifier to check that untrusted binary modules will maintain a range of restrictions on the dynamic execution state and that software guards are present where constraints cannot be checked statically (for example on computed branches). A binary translator is used to generate such modules from existing binaries but does not form part of the trusted computing base. XFI uses control flow integrity (CFI) [57] based on guards at computed call sites which check for valid identifiers stored just before valid branch targets. Two separate stacks are used to ensure that critical variables, such as the stack pointer and return address cannot be corrupted: the scoped stack has a known constant frame size at all points in a function and is only accessed statically, whilst the allocation stack is used for variables whose address is taken. Guards for those memory accesses which cannot be verified statically have a fast-path for when the access lies within an expected region and a slow-path for more complex cases. The checks can be shared between multiple accesses where CFI ensures that the check will always be performed before any of them takes place.

The XFI binary translator is, by the authors' own admission, both complex and slow and lacks support for commonly used extensions such as the MMX and SSE vector instructions. They claim that the verifier, on the other hand, is conservative, simple, and fast, partly because it relies on (untrusted) hints generated by the translator.

Vx32 [50] and Google's native client [16] (NaCl), which encapsulates untrusted native code downloaded from websites, employ a hybrid technique which combine static validation with use of the segment registers available on x86 processors. The use of extra hardware support allows them to remove the need for dedicated registers (critical on the register poor x86 architecture) and add support for confidentiality, but restricts NaCl applications to 32-bit compatibility mode on modern 64-bit processors because the little-used segment registers were removed from 64-bit x86. This later proved to be a problem for Google as performance of this legacy feature on Intel's low power Atom processors was so poor that the developers had to seek an alternative approach.

Software fault isolation (SFI) has performance advantages over TLB and virtual machine based methods but it can introduce onerous restrictions on sandboxed code and often has considerable performance overhead. Implementations differ but common limitations include: protection for memory writes only; a single data region per sandbox necessitating copying of argument data during domain crossing; or the assumption of a trusted parent/untrusted sandbox model with no support for inter-sandbox calls. The case of NaCl in particular indicates a frustrated need for hardware support for compartmentalisation.

2.3.10 Hardware compartmentalisation support – address based

The need for fine-grained memory protection and compartmentalisation has long been recognised by hardware designers, but as yet none of the proposed technologies have gained widespread adoption, probably due to the difficulty of maintaining backwards compatibility and an incremental adoption path, along with the perception that page-level protection is 'good enough'. Hardware compartmentalisation schemes can be broadly categorised as address based or pointer based, depending on whether protection information is stored on a per virtual address or per pointer basis. Address-based approaches

typically augment or replace traditional page table structures with similar, potentially fine-grained, permissions tables and provide a means for security domain switches within a single address space.

Protection Lookaside Buffers

Koldinger et al. [58] consider two table-based protection architectures in the context of a single address space operating system, hypothesising that security domain switching will become increasingly common on such systems. Firstly they suggest decoupling virtual address translation from protection using a Protection Lookaside Buffer (PLB) that operates similarly to a TLB except that entries do not contain translation information. When coupled with a virtually-indexed, virtually-tagged cache this has the advantage that the TLB can be removed from the critical path for memory access, and also allows the page size of the PLB and TLB to be independently optimised for protection granularity and virtualisation efficiency respectively. Sharing between security domains still requires one entry per process in the PLB, but TLB entries are shared due to the single-address space design. In theory this allows the PLB to have a greater capacity than the equivalent TLB because the entries are smaller, however the space savings are minimal since only the physical address is saved and physical addresses are typically smaller than virtual addresses [59]. Moreover these advantages only apply to single address space operating systems, which have not been widely deployed despite the widespread availability of 64-bit hardware. A major reason for this is backwards compatibility: existing software often makes assumptions about address space layout and would need significant alteration to run in single address space systems.

Page groups

Koldinger also discusses the use of page groups as implemented on the Hewlett-Packard PA-RISC, where TLB entries are tagged with an *access ID* instead of a process ID. The *access ID* is then checked against the set of allowed page groups for the current process to determine whether access is permitted (only four active page group registers were available on the PA-RISC rising to eight on later versions, whereas Koldinger consider a hypothetical larger, hardware-managed LRU cache). This arrangement means that sets of pages can be shared between processes without duplicate entries in the TLB, but requires an extra hardware stage after TLB lookup to check the access permissions and a limited, associative resource – the page group registers – which must be flushed on every security domain switch. Although PA-RISC allows page groups to be set read-only per process, permissions are otherwise the same for all processes accessing the page group limiting the range of possible sharing arrangements.

Mondrian Memory Protection

Mondrian Memory Protection [17] [18] (MMP) adopts the idea of a protection table distinct from the translation table to provide fine-grained memory (word level) protection and multiple domains within a single address space. The permissions are stored in memory in a complex, multilevel, compressed format which is cached in a PLB structure and in *side-cars* attached to each general purpose register. The proposals also extend to a fine-grained memory translation mechanism, which the authors use to implement a zero-copy

networking stack. Domain transitions are achieved using specially tagged protection table entries which indicate that an address contains a *call gate* to a given protection domain ID. On branching to such a location the processor automatically pushes some minimal context to a special *cross domain call stack* and switches to the called domain. A later call to a *return gate* results in the reverse procedure, popping the stack and returning to the caller's domain. Witchel and Asanović ran a range of benchmarks and made a version of Linux modified to use MMP to separate modules within the kernel address space. The benchmarks showed moderate memory usage overhead (11%) and about 12% performance overhead based on simulation, but there is no attempt to build a hardware implementation or evaluate hardware complexity.

One limitation of MMP is that elevated privileges are required when modifying permission tables, leading to a domain transition or supervisor call for every memory allocation or permissions change. For very fine-grained memory protection this could be a significant overhead, limiting the granularity achievable. Furthermore on multicore hardware the PLB and side-cars of all cores will need to be flushed on every permission table modification, potentially leading to heavy performance penalties similar to that incurred by TLB shootdowns. Alternatively a coherence mechanism could be provided to keep the cached permissions synchronised, at the cost of some additional hardware complexity.

Pointer-based memory protection schemes as explored in the following sections can avoid both of these problems inherent in table based protection, but may suffer from other problems such as greater difficulty revoking privileges which have been delegated.

2.3.11 Hardware compartmentalisation support – pointer based

Whilst address based compartmentalisation techniques seem like a natural extension of existing TLB structures they require supervisor intervention for the creation and maintenance of compartment protection information, limiting their performance and scalability. Secondly, associating protection information with addresses risks permitting access when a pointer moves outside of the intended object but still falls within another valid object (as with a buffer overflow).

As an alternative, protection information can be associated with the pointers (references) rather than with memory regions. Storing protection information once per reference rather than per object is potentially costly in terms of memory usage, but affords extra flexibility in that different pointers to the same object can have different permissions, and is stricter in that a pointer to one object cannot be used to inadvertently access another object in the same address space. Pointer protection information can either be stored inline with each reference (also known as *fat-pointers*) or in an indirect pointer table indexed by a pointer ID. Inline bounds storage has the disadvantage of inflating the size of pointers, which has a potentially large impact on cache and memory usage especially in pointer heavy code, but the latter requires an extra translation stage on each memory access which might increase latency.

Pointer based schemes mean that objects can be shared in a natural fashion by passing pointers without the need to copy data or adjust permissions tables to set up sharing, leading to greater performance and ease of use.

Capability systems

The Burroughs B5000 [60, 61] from the early 1960s was the first machine to add hardware support for bounded pointers with with its *descriptor table*, which held 1024 entries containing either values or *descriptors* pointing to *segments* (bounded, contiguous memory regions). Each entry had a type *tag* to prevent accidental dereferencing of data values or execution of data segments, and all memory accesses were checked against the segment bounds. This proved a very influential design and later machines adopted many of its features.

The term capability was first used by Dennis and Horn in their seminal 1966 paper [62] which describes a series of abstract operations for use in a 'multiprogrammed' system (i.e. any system capable of running more than one program simultaneously, either through time slicing or multiple CPUs). Central to this was protecting programs from each other by placing them within their own *spheres of protection*, for which they envisaged the use of a per process *capability list* (C-list) containing an array of *capabilities*. Each capability grants access to some object (such as a file or region of memory) with given *access indicators* (permissions such as read, write or execute), and access to every object is via an index into the capability list, allowing the permissions to be checked. The owner of a capability can *grant* use of it to another process by adding it to its C-list, specifying the permissions to be delegated and later *ungrant* by removing it from the list. Crucially, they also describe *protected entry points* which allow secure transitions between processes using *entry capabilities*, which are special capabilities granting permission to branch to a specified entry point in another process whilst simultaneously loading the corresponding C-list and copying a single argument capability.

The provably secure operating system (PSOS) [63–67] developed at the Stanford Research Institute went on to provide a formally proven basis for the security of capability systems, and the ideas were adopted by a number of research and commercial computer systems, each differing in their interpretation and implementation [60]. Notably the Plessy System 250 [68], designed to operate a highly reliable telephone switching system, was the first practical system to use capabilities and the Cambridge CAP Computer [69] was the first developed successfully in an academic context. Both of these used capability registers to store capabilities for immediate manipulation and allowed them to be stored in memory in dedicated capability segments (separate from the data segments to prevent unauthorised modification of capabilities). They also featured protected procedure calls with arguments being passed via capability registers.

Hydra [70], developed at Carnegie-Mellon University, was an experimental operating system which used a software implementation of capabilities to provide abstract *objects* to the programmer. Objects had a *name*, which served to uniquely identify them, a *type*, which defined the permitted operations and a *representation*, which contained the data associated with the object, including a list of capabilities to other objects. A *type manager* was responsible for the creation and manipulation of objects of a given type. To support this Hydra allowed the creation of *restricted* capabilities which identified an object but did not permit direct access to data. These could be used by the holder of an object as a token and later passed back to the type manager, which could then *amplify* the capability to gain access to the data. However, since the Hydra was implemented in software on top of generic PDP-11 minicomputers it had some limitations, such as that all object manipulation had to occur via the kernel and involved copying of data to and from local memory.

The IBM System/38 [71] was a major commercial capability system which added hardware support for objects, but aimed to separate the programmer from the implementation through the use of microcode to implement high level capability operations in terms of more efficient low-level machine instructions. This proved an effective strategy and System/38 programs are still being supported on current IBM Power systems via emulation. An interesting feature is the use of *tagged memory* which allows capabilities and ordinary data to be mixed in the same memory regions: each memory word is associated with an extra *tag* bit indicating whether it contains a capability or not. These bits, which are not accessible to the programmer, are set when capabilities are stored and cleared when data is stored, ensuring that capabilities cannot be modified in memory to grant unauthorised access.

The Intel iAPX 432 [72] (Advanced Programming arCHItecture) was another major commercial deployment of capabilities which featured sophisticated support for objects where all data could be treated as an object. Some types had hardware support and others were handled transparently via software, making for a very consistent programming model. It even had hardware support for garbage collection in the form of bits in the capability descriptor to support the mark-and-sweep algorithm. To address the problem of protecting capabilities in memory the Intel 432 allowed memory segments to be split into two, with one exclusively for capabilities and the other for data. This means that only one data segment was required per object but does not provide such a natural data layout as tagged memory. Despite its relative sophistication the Intel 432 was a commercial failure. This was due to the poor performance of the initial implementation owing to ambitiousness of the project and an unoptimised compiler offering, which meant that it was outperformed by other, cheaper processors available at the time.

With the RISC movement [73], which focused on simplifying instruction sets to reduce hardware cost and complexity whilst providing an efficient compiler target, interest in capability systems waned. The capability model was seen as too complex to implement in these simplified machines, which adopted the much simpler paged virtual memory approach used today.

Nevertheless capability systems continue to hold an attraction due to their support for fine-grained memory protection, clean mapping to object-oriented programming models and effective support for compartmentalisation with efficient data sharing and domain transitions. The increasingly apparent need for better computer security and the massive decrease in the cost of transistors due to improved fabrication technology has led to renewed interest in the field, and a number of research projects have sought to create modern capability systems without violating the principles of RISC design.

M-Machine

The experimental M-Machine architecture [19] supported *guarded pointers*, which include protection information in the unused upper bits of a 64-bit pointer (only 54-bits of virtual address space were supported). It used a simple scheme where segments must be power-of-two sized and aligned, meaning that the bounds can be specified using just the 54-bit offset and a 6-bit exponent that gives the length of the segment as a base-2 logarithm. There is also a four-bit permissions field which includes read, write, execute, 'enter' privileges and a single *tag* bit that indicates the presence of a pointer in a register. M-machine enforces restrictions on pointer manipulation: special instructions for pointer arithmetic throw an exception if the result is out-of-bounds; normal arithmetic instructions clear the tag in the

result; loads, stores and jumps require a valid pointer operand for the address. Pointer manipulation does not require special privileges creating a safe, efficient way for user-mode code to restrict pointers for its own use and for passing between security domains.

M-machine also supports 'key' pointers, where the permissions indicate that a pointer cannot be modified or used for memory access. Such 'sealed' pointers can be 'unsealed' by privileged software, making them useful as unforgeable tokens or identifiers. Similarly, 'enter' pointers act as minimal primitive for transitioning to a new domain via a call gate: they cannot be used or modified except by jumping to them, at which point control transfers to the fixed entry point specified by the pointer, which is simultaneously unsealed and made available via the program counter. Any data pointers required by the new domain must be stored inline with the code or in a thunk, so that they may be accessed via the instruction pointer after the call.

M-machine uses *tagged memory* to preserve the tag bit when pointers are loaded and stored requiring one extra bit of storage per 64-bit word but ensuring that pointers are unforgeable, meaning that guarded pointers are *capabilities*. Privileged code has the ability to fabricate arbitrary pointers via the `SETPTR` instruction so the M-machine cannot be used to compartmentalise privileged code, however the amount of code which needs to execute in privileged mode can be greatly reduced through the use of guarded pointers and protected subsystems.

The M-machine achieves memory efficiency using a compressed capability format which fits in the same space as a standard 64-bit pointer, but this comes with two significant costs: firstly, objects can only be allocated in power-of-two sized and aligned blocks which could lead to inefficient memory use due to fragmentation (nearly 100% for objects just over a power of two). Secondly, all valid pointers are implicitly within bounds, with code which causes the pointer to move out-of-bounds triggering an exception. This makes it impossible to implement a compatible C-compiler with support for common C idioms such as the *invalid intermediate* pattern identified by Chisnall et al. [74]. The exception cannot be delayed until dereferencing because there is no way to represent an out-of-bounds pointer if it has to be stored in memory, such as to spill a local variable to the stack.

SAFE Processor / Low-fat pointers

More recently Kwon et al. [75] describe an interesting scheme using compressed, fat-pointers which are tagged with a type for unforgeability, similar in many ways to M-Machine's guarded pointers. The compression scheme allows fat-pointers into a 46-bit virtual address space to fit in a 64-bit word, and is designed to be efficient to implement in hardware whilst minimising memory fragmentation needed for padding (3% worst case). There is limited support for out-of-bounds pointers by allocating inexact regions and filling with poison words tagged with *out-of-bounds-memory-location*, but no support for many of the other C-idioms identified by Chisnall et al. such as *Mask*. In the extended SAFE processor [76], domain transitions are achieved through call gates which consist of sealed code, data and *authority* triples, where the authority is a typed-pointer to a structure which describes the privileges of the currently running process.

PICA

PICA [77, 78] is a capability machine which extends a MIPS32 processor with a set of dedicated capability registers that refer to a memory-resident capability mapping table. All memory references occur via capabilities, which act as a level of indirection replacing the traditional page table with one indexed by capability ID instead of virtual address. Translation caches are used to accelerate this. Tagged memory protects capabilities when they are stored in memory. Sealed *entry capabilities* that only allow branching to a given fixed entry point are used for domain crossing purposes. Domain crossing involves a sequence of several instructions but to ensure that they occur in the correct order a hardware *sequence tracker* throws an exception if they are used incorrectly. PICA provides two modes for managing sandbox *contexts* (stacks): in the 'independent contexts' case a separate context object is used for each new cross-domain call, but to avoid the overhead of allocating a new context for each call the hardware supports managing a pool of previously allocated contexts. Special instructions manage lists of free and used contexts so that they can cleared in the background asynchronously, but in addition there is an instruction for efficiently clearing contexts using a *high water mark* which records the maximum extent of stack used. The unified context mode is implemented by enforcing an additional lower bound on the current context, preventing a callee from accessing the caller's stack. The stack bounds and stack pointers are pushed and popped to a secure, hardware-managed stack during domain crossing. Finally a *mask* instruction allows for clearing multiple registers efficiently during domain transitions.

The authors found that the overheads of domain crossing were moderate at 15-20% based on simulation results, and that using independent contexts was considerably more expensive than a single unified context though little explanation is given for this. No hardware was actually built so it is difficult to know whether the authors' assertions about modest hardware implementation costs are realistic, in particular the `PCALL` instruction requires updates to many bits of CPU state including every register in the capability register file. Moreover the abandonment of paged virtual memory would pose a challenge when porting existing operating systems.

CHERI

CHERI is a capability extension to the MIPS ISA, but rather than using capabilities containing a unique ID referencing an in-memory translation table as did PICA, the bounds are stored directly in the capability data type in the form of fat-pointers. This makes it possible to retain the existing TLB for virtual address translation, improving backwards compatibility with existing operating systems. As it is the main subject of this dissertation, CHERI is described in more detail in Chapter 3.

CODOMS

CODOMS [21] (Code-centric memory domains) is another take on address space compartmentalisation based on the observation that there is a strong relation between a domain's code and the data it can access. Vilanova et al. propose associating a protection domain ID with each page in the existing TLB and describe *access protection lists* APLs which specify the access rights (read, write and call) of each domain to the pages of others. The domain ID of the page corresponding to the program counter is used as the current domain ID (hence code-centric domains). This results in a similar protection to page

groups except that the current set of page groups is determined by the current PC. Domain switches are a achieved by tagging certain pages with 'call' permissions which allow branching from one domain into another via a fixed entry point in the page.

To aid with passing arguments between domains CODOMS layers a capability scheme on top of the page based protection. Domains can create capabilities to arbitrary byte-granularity subsets of their address space and pass them via special registers to callees. The creator of a capability's APL is copied into the capability such that users of the capability are delegated the same rights, though the permissions granted can be further restricted by user level instructions. CODOMS also distinguishes 'synchronous' from 'asynchronous' capabilities with the former having the restriction that they cannot be stored to memory and are thus suitable for delegating rights just for the duration of a synchronous call and return (provision is made for spilling them to hardware controlled 'domain control stack' which ensures that they are revoked on return). Asynchronous capabilities are intended to be longer lived and include a pointer to a counter in memory which can be used by the originator of the capability to revoke all capabilities which reference that counter.

CODOMS is a hybrid of address-based and pointer-based protection which attempts to provide the advantages of both methods. Unfortunately the use of the PC to determine the current domain restricts each sandbox to a single instance, limiting the granularity of sandboxing. This is further compounded by the use of TLB pages as the basic unit of compartmentalisation, although byte-granularity capabilities for argument passing mitigates this somewhat.

Special purpose hardware compartmentalisation

Recent efforts to provide a degree of compartmentalisation in hardware such as ARM's TrustZone [79, 80], Iso-X [20] and Intel Security Guard Extensions (SGX) [81] are significant, but limited in flexibility and scope. Specifically, TrustZone virtualises a mobile processor and introduces a reference monitor in order to allow a small, trusted operating system (the *secure OS*) to share a CPU with an untrusted, commodity operating system, but this only benefits the applications running on the secure OS and is inherently limited to a single secure zone. Similarly Iso-X and SGX aim to provide protection for critical applications against malicious system software or even physical attacks by providing special hardware enforced encrypted compartments. They use cryptographic hashing to attest to the integrity of the compartment and encryption of memory to preserve the confidentiality of computation inside. However, they rely on the page table for isolation, limiting compartment granularity, and can only be used to compartmentalise *trusted* applications (such as Digital Rights Management or authentication protocols). They are useful, but have a fundamentally different threat model to compartmentalisation solutions which aim to protect applications from parts of themselves (in the form of untrusted extensions or programmer errors leading to vulnerabilities). They are, therefore, not directly comparable to CHERI.

Memory safety without compartmentalisation

Some proposals such as Hardbound [82] and Intel's new Memory Protection Extensions (iMPX) [83, 84] provide fine-grained bounds checking of memory accesses but are not suitable for compartmentalisation. Hardbound and iMPX place bounds information about

each pointer into a separate table in memory and include hardware support for bounds checking on access (either enforced or, in the case of iMPX voluntarily using extra instructions). However because the bounds table is modifiable without restriction, pointers can be easily forged making these unsuitable for containing sandboxed code.

2.4 Summary of Compartmentalisation Methods

Memory protection and compartmentalisation in computer systems has a long history and a wide variety of different schemes have been proposed and implemented. In the early days sophisticated architectures based around capabilities and objects were developed, but with the advent of RISC computing these were discarded in favour of simpler and cheaper paged virtual memory. This left the essential task of compartmentalising up to software, either in the form of process based isolation or high-level languages. Unfortunately process based isolation suffers from high performance overhead due to TLB constraints and high-level languages have large TCBs and are unable to support the significant bodies of legacy code. They proved adequate for some time, but as awareness of the need for better security became more apparent programmers were forced to improvise ever more elaborate schemes, such as SFI and its derivatives, to work around the inadequate hardware support.

Mondrian was one hardware response, but found it difficult conceptually to move beyond the notion of the page table structure and its associated disadvantages in terms of pointer-safety and scalability to multicore architectures. Re-inventions of capability mechanisms along RISC principles such as M-Machine, SAFE, PICA and CHERI show greater promise, providing an attractive combination of pointer-safety and a clean mapping to object-oriented programming styles for fine-grained, within address-space compartmentalisation. However, they have not all paid enough attention to legacy support, with CHERI being the only one of these with proven support for a large existing C-language operating system and an incremental path for adoption of protection features.

CHAPTER 3

CHERI

3.1 The CHERI ISA

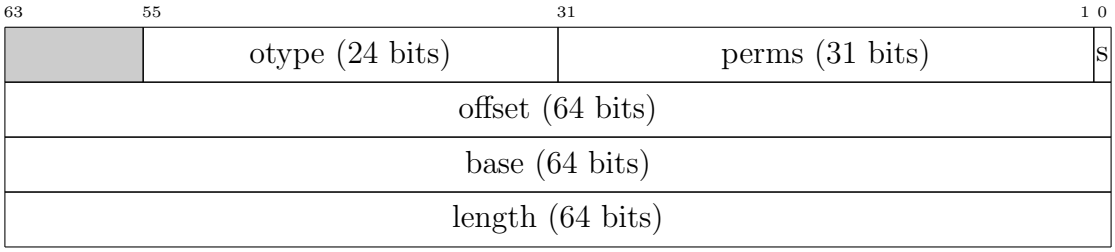
CHERI (Capability Hardware Enhanced RISC Instructions) is an experimental extension to the 64-bit MIPS ISA designed to add memory safety features using *capabilities* [24, 25]. In this chapter I describe these features and how they can be used to efficiently compartmentalise applications within a single virtual address space.

3.1.1 Memory Capabilities

In the context of computer security a capability is defined as a communicable, unforgeable token of authority [62]. On CHERI these take the form of 256-bit *fat pointers* whose format is shown in Figure 3.1. These are traditional memory pointers augmented with a base address, length and permissions that specify, with byte level granularity, the limits within which the pointer (offset) is valid. Capabilities in the CHERI ISA are a first class *pointer* datatype, on a par with integers or floating point numbers in traditional MIPS. They are stored in 32 special registers, and can be loaded and stored to memory just like ordinary values, except that when in memory they are identified by a tag bit which prevents unauthorised modification (see Section 3.1.3). CHERI defines a set of unprivileged instructions for safely manipulating capabilities such that the access granted can only be decreased. Load and store instructions are provided to access memory via capabilities and for loading and storing capabilities themselves . The offset field in the capability plays the role of a pointer that is added to the immediate from the load or store instruction to form an *effective address*. This offset is allowed to stray out-of-bounds so as to permit behaviour exhibited in real world C programs [74], but permissions and bounds checks are performed on memory accesses – the effective address must be within bounds or an exception will be raised by the hardware. Capabilities use virtual addresses: after bounds checks the effective address is translated via the TLB to compute the physical address to use.

CHERI capabilities have all the memory safety benefits of the *guarded pointers* used on the M-Machine [19], but are also a practical compiler target, allowing pointers to stray out-of-bounds when required and supporting the idioms used by real C programs [74]. Their main disadvantage is their size and consequent memory footprint, but previous work has shown that the overhead is acceptable in benchmarks [22] and ongoing work

Figure 3.1: Fields of a 256-bit CHERI capability



to compress capabilities could reduce them to 128-bits whilst maintaining their benefits. In Section 4.4 I evaluate the effect of saving and restoring the capability register file on kernel `syscall` performance.

3.1.2 Backwards Compatibility

For backwards compatibility with legacy MIPS load and store instructions, which do not specify a capability register to use, CHERI uses the *default data capability*, `$c0` also known as `$ddc`. By default this encompasses the entire address space but it can be restricted in order to form a compartment (see Section 3.1.4). Similarly, the program counter (PC) is indirected via the special *program counter capability*, `$pcc`, which can only be modified using a capability jump instruction (`cjal`), or when entering and leaving the exception handler.

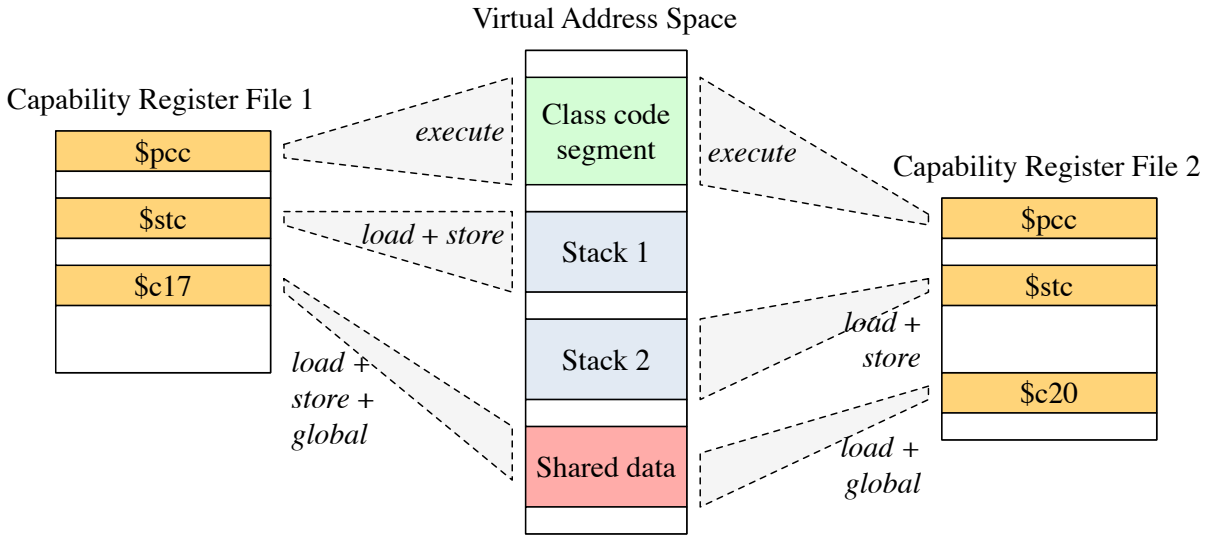
3.1.3 Tagged Memory

To protect the integrity of capabilities when they are in memory and maintain their unforgeability CHERI uses *tagged memory*. For every 256-bit aligned memory location CHERI maintains a single tag bit that indicates whether that location holds a valid capability. The bit is set when a valid capability is stored and cleared on any write of non-capability data to that location. A tag controller stores these bits in a dedicated portion of memory that is not accessible via normal memory operations, requiring a memory overhead of $1/256 \approx 0.4\%$. To reduce the number of off-chip memory requests the tag bits are propagated and cached at each level of the memory subsystem, including the L1 and L2 caches and a dedicated tag cache in the tag controller.

3.1.4 Capabilities for compartmentalisation

Since all memory accesses on CHERI are constrained by unforgeable capabilities, the set of capabilities available to a thread determines its accessible address space. This set is formed of the contents of the capability register file (including the default capability, `$c0`), the program counter capability `$pcc`, and any capabilities stored in memory accessible via these capabilities (including recursively by indirection). Thus threads can be isolated from each other by restricting their capability contexts, allowing for fine-grained compartments within the same address space as shown in Figure 3.2. Performing a domain transition then means simply switching between threads, with no need to manipulate the TLB and hence no call into the privileged kernel to do so. However, for true isolation there must still be some mechanism to securely restore the new thread's context without giving the

Figure 3.2: CHERI allows a single virtual address space to be divided into multiple parts using capabilities. Here two compartment threads are shown sharing a code segment and some data, but have separate stacks. Figure courtesy R. Watson [23].



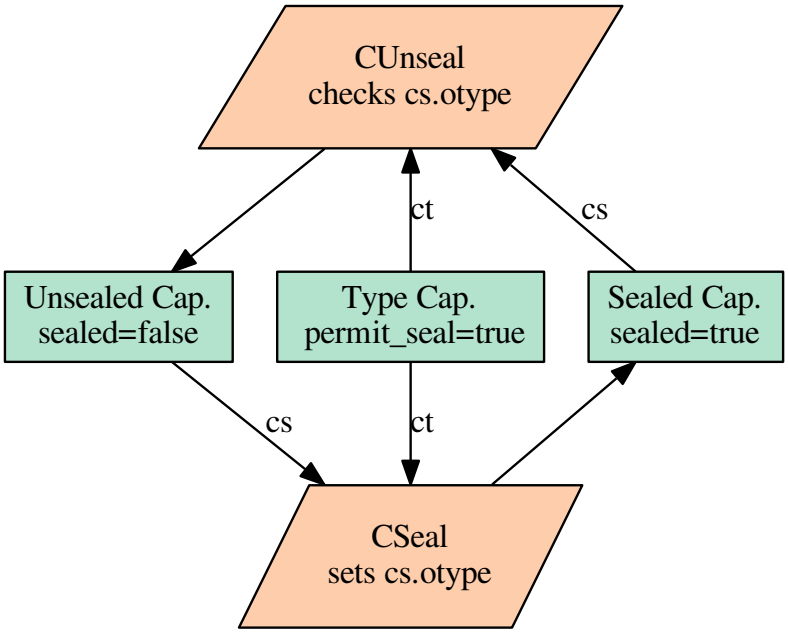
caller direct access to it. In the absence of multiple hardware threads this requires some form of privilege escalation or call gateway (such as a system call). FreeBSD's lightweight processes (threads) are sufficient for this, however as shown in Section 4.5 the overhead of thread switching is significant, and requiring each compartment to be its own thread introduces unnecessary parallelism which complicates the job of the programmer.

3.1.5 Sealed capabilities

CHERI supplements its memory protection features with *sealed capabilities*, which can be used as transferable tokens of authority that do not grant memory access, serving as a foundation for the implementation of protected entry points described in Section 3.1.6. They are similar to M-Machine's 'key' pointers but unlike on M-Machine unprivileged code can perform the unsealing provided it has an appropriate capability granting it the authority to do so. This is implemented using a 24-bit **otype** (object type) field and a **sealed** bit in the capability format and two instructions, `CSeal` and `CUnseal`, as shown in Figure 3.3.

The `CSeal` instruction takes two capability operands: a source, `$cs`, and a type, `$ct`, which must have the `Permit_Seal` permission. It seals the source capability by setting `$cs`.**sealed** to one and `$cs`.**otype** to `$ct`.**base** + `$ct`.**offset** (note that the **otype** must be within the address range of the type capability, meaning that capabilities with `Permit_Seal` grant the right to seal with a limited range of types, and that these rights can be delegated just like other capabilities). A sealed capability cannot be modified or used for memory access, but can be passed in registers or stored and loaded in memory. Sealed capabilities can then be unsealed using the `CUnseal` instruction and a *matching type capability*. Thus, sealed capabilities can be used as tokens which can be passed to an untrusted entity and later retrieved and unsealed. For example this mechanism can be used to enforce the idiom of opaque pointer types commonly found in C programs, ensuring that only the code which creates an object can manipulate the data contained within it, but allowing

Figure 3.3: Capability sealing: Green rectangles represent capabilities, red parallelograms are instructions. Capabilities can be *sealed* using another capability as a key. Sealed capabilities do not grant memory access privileges and cannot by modified, but can be passed as tokens which can be unsealed later using the key. This can be used to enforce the *opaque pointer* idiom common in C programs, and also acts as the foundation for the `CCall` object invocation instruction.



references to be passed outside and later returned for processing.

3.1.6 Domain crossing: `CCall`

For sealed capabilities to be effective the 'type' capability used as a key must be stored in a separate security domain to prevent users of sealed capabilities from gaining access to it, unsealing their capabilities and performing arbitrary manipulation of the data. Using Hydra terminology this domain would be known as a *type manager* and would have sole responsibility for creating and manipulating object instances of a given type; using more familiar object-oriented terminology the domain might be called a *class* and individual operations *methods*.

Under this paradigm every operation on a sealed object requires a transition into another security domain and back which, as shown in Section 4.3, would incur significant overhead using standard asynchronous inter-process communication techniques for domain crossing. To address this CHERI includes the `CCall` instruction which can be used to accelerate domain crossing by providing a form of direct, synchronous invocation on sealed 'objects' consisting of a pair of capabilities pointing to the object code and data. `CCall` uses a sealed executable capability to implement a call gate in a similar way to M-Machine's 'enter' pointers, or 'entry' capabilities of SAFE and PICA: it permits branching to a safe entry point whilst simultaneously elevating privilege by unsealing capabilities. This direct, synchronous transfer of control operates more like a function or method call than traditional asynchronous domain crossing techniques, simplifying the programming model and reducing overhead.

Unlike on M-Machine the capabilities that represent the code and data for an object are bound together by the **otype** field. This removes the need to store data or key capabilities inline with the code, allowing multiple instances of a class to share code without the use of per-object thunks. It also ensures that the class code can only be invoked with a data capability of the correct type, which helps to address the 'confused deputy' problem (where privileged code is tricked into abusing its authority, for example by passing it an argument of the wrong type).

In detail, the `CCall` instruction requires two sealed capabilities with matching types as operands: the code and data capabilities, `$cc` and `$cd` (the former having the `Permit_Execute` permission and the latter lacking it). Figure 3.4 shows how `CSeal` and `CCall` can be used to construct object capabilities and invoke a sandbox. Invoking `CCall` atomically checks the **otype** fields of `$cc` and `$cd` match, unseals them and branches to `$cc`, creating a call gate which can be used to perform secure domain transitions. It also pushes the current `$pcc` and *invoked data capability* (`$idc`) to a special *trusted system stack* and overwrites them with the unsealed contents of `$cc` and `$cd`.

The trusted stack is a special per-thread stack only accessible within the trusted handler, and acts as secure place to store a pointer to the caller's context to restore on `CReturn` (Figure 3.5). The use of a stack structure for this enforces strict call-and-return semantics, meaning that other calling behaviours such as tail calls or continuation passing are not supported. This has the advantage that it is possible to recover from a failed `CCall` (for example if a sandbox enters an infinite loop) by unwinding the stack, but does not exclude implementing more exotic control flow in the future. In principle `CCall` could be implemented purely in hardware or using microcode, but for flexibility and ease of experimentation the current CHERI prototype uses a trap instruction which branches to

Figure 3.4: `CCall` diagram. Green rectangles represent capabilities, red parallelograms are instructions. We start with code, data and type capabilities then seal the code and data capabilities using the type capability. The sealed capabilities cannot be modified or used to access memory so can be safely passed to untrusted code, which can then use them to make a `CCall`. After the `CCall` the code capability is installed in `$pcc`, the unsealed data capability is placed in `$idc`, and the `$pc` is set to the value of the offset field from the code capability. This means that control passes to the trusted *call gate* when the data capability is unsealed, enforcing only legitimate access to the object.

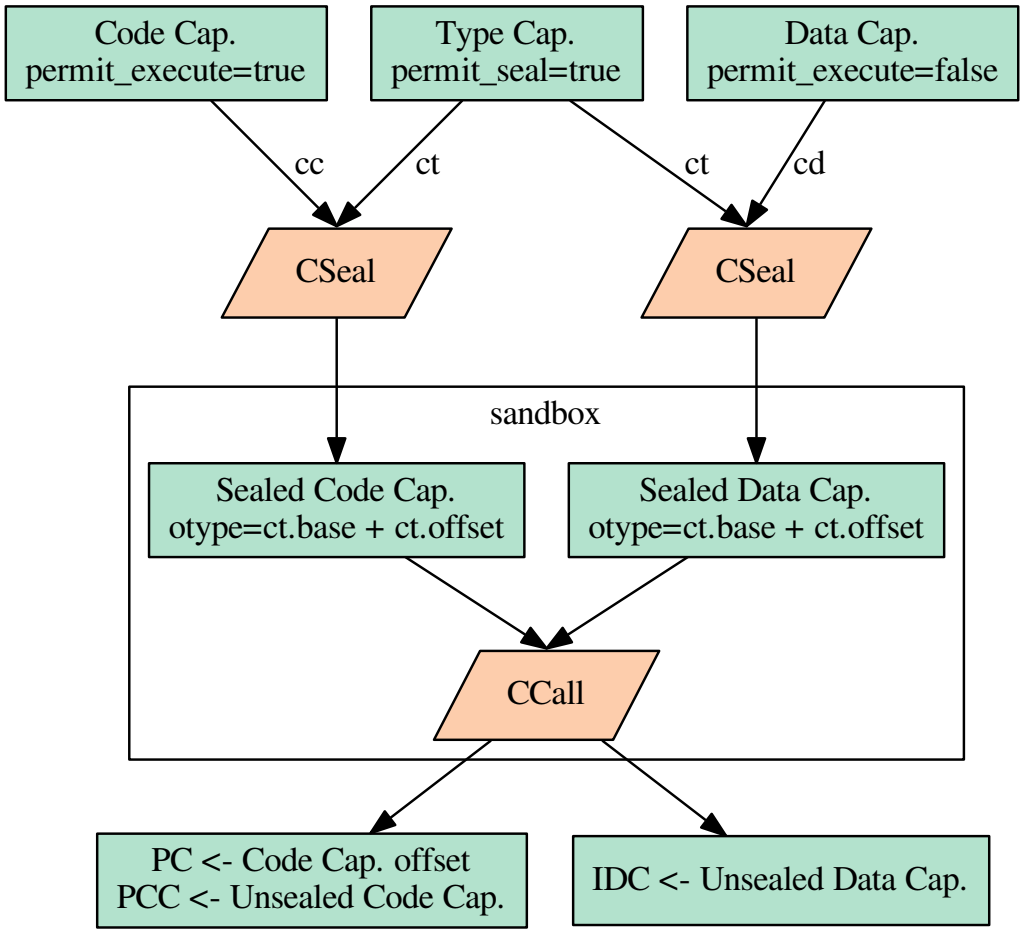
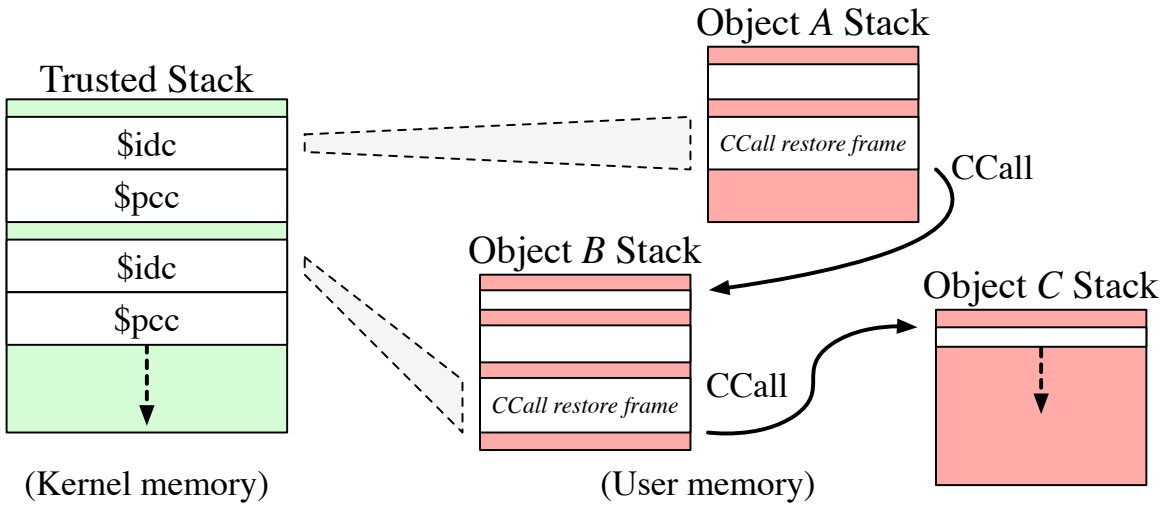


Figure 3.5: The trusted stack resides in per-thread kernel allocated memory and acts as a secure location to store a pointer to the caller's frame to be restored on `CReturn`. Figure courtesy R. Watson [23].



a small handler running in kernel mode. This is analogous to the way MIPS features a software TLB miss handler instead of a hardware page table walker. `CReturn` is the converse instruction to `CCall`, which simply pops `$pcc` and `$idc` from the trusted stack, returning control to the caller. Like `CCall`, `CReturn` is implemented as a trap on CHERI.

`CCall` and `CReturn` provide the foundations for capability-based domain transitions, but are not sufficient on their own; rather a complete capability invoke requires the co-operation of the compiler, a user-space library and a kernel trap handler. First, the caller must save its own state such that it can be restored on return and clear all registers (both general purpose and capability) whose contents it does not wish the caller to have access to (i.e. all registers that do not contain arguments for the call). The caller saves most of its state to its own stack and prepares a minimal data frame containing the stack pointer, stack capability and the default data capability, pointed to by `$idc` which will be saved to the trusted stack by the `CCall`. Each compartment has its own stack which must be installed before calling the compartment's code. Before calling `CReturn` the compartment must clear its register state to avoid leaking it to the caller. [1] On return `$idc` is popped from the secure stack, allowing the caller to restore its stack and register state from the previously constructed call frame.

These tasks are divided between the compiler, a user-space handler (`libcheri`) and the trap handler according to practical, performance and security considerations as summarised in Table 3.1. The compiler is responsible for clearing unused argument registers and saving the caller-save registers because only it knows which of these registers are in use. The caller must save all callee-save state and prepare the `$idc` pointer on every call, so this code can be can be shared between all call points by placing it into an unprivileged library, called `libcheri`, which also restores state on return. The trap handler must perform the trusted stack manipulation as only it executes with the required privilege. It also performs checks on the capability argument and return registers to ensure that *local* capabilities do not cross security domain boundaries (see Section 3.1.7). Finally,

[1] The current sandboxing model used by CheriBSD assumes mutual distrust between security domains – if one side of the transition trusts the other then some register clearing can be avoided.

Table 3.1: `CCall` and `CReturn` stages. Some parts of the `CCall` are performed by the compiler at the call point, others by the user space `libcheri` library and others by the `CCall` trap handler. Those parts which are needed in order to provide the desired function call semantics but are not security critical are marked ABI (Application Binary Interface).

Action	For	Responsibility	Reasoning
Save caller-save state	ABI	compiler	Has info.
Clear unused arguments	Security	compiler	Has info.
Save callee-save state	ABI	`libcheri`	Code reuse, no priv.
Bundle sp/IDC	ABI	`libcheri`	Code reuse, no priv.
Validate `CCall` args	Security	`CCall` handler	Privileged
Push trusted stack	Security	`CCall` handler	Privileged
Clear non-arg state	Security	`CCall` handler	Code reuse
invoked code			
Clear unused return state	Security	compiler	Has info.
Clear non-arg state	Security	`CCall` handler	Code reuse
pop trusted stack	Security	`CCall` handler	Privileged
validate return value	Security	`CCall` handler	Privileged
Unbundle sp/IDC	ABI	`libcheri`	Code reuse, no priv.
restore callee save state	ABI	`libcheri`	Code reuse, no priv.
restore caller save state	ABI	compiler	Has info.

the kernel handler clears all non-argument registers to ensure no accidental leakage of information or capabilities between domains. This clearing could take place in user space but doing it in the kernel handler gives the most code re-use and ensures that it occurs on every domain transition. If it were left to user code there is the possibility that it may accidentally be omitted and it also removes the need to defensively clear registers on the receiving end of a transition since both sides trust the trap handler.

3.1.7 Local Capabilities

In common with other capability systems, CHERI has the problem that once a capability has been granted it is difficult to later revoke that privilege, for example in order to reuse a region of memory for an object which is no longer needed. The problem is that all references to the object must be located and invalidated, a process which might involve searching through all memory where a reference might have been stored (e.g. using a garbage collector). This is known as *temporal* safety as opposed to the *spatial* safety naturally enforced by CHERI capabilities. On CHERI it is possible to avoid searching by unmapping the TLB pages where the object is located, but that may be unsatisfactory as it does not allow deallocation of regions smaller than one page, eliminating the benefits of fine-grained memory protection, and also prevents reallocation of that virtual address space which is undesirable even though virtual address space is less precious than physical memory.

CHERI attempts to address this with *local* capabilities, which are simply capabilities with the *global* permission bit unset. Local capabilities cannot be stored to memory except through capabilities which have the **store-local** permission.

This feature allows software to restrict the places where local capabilities can be stored, and hence which regions need to be searched to revoke a capability (in effect, creating a single bit information flow control policy). Local capabilities have been used in two different ways during the evolution of the CHERI ecosystem:

1. In the initial implementation of sandboxing in CheriBSD only local capabilities were passed across domain boundaries, and only the stack had the **store-local** permission required to store them in memory. This would allow references to be passed to a sandbox for the duration of a call and spilled to the stack if necessary. On return from a `CCall` the caller would only have to clear the sandbox stack to ensure there was no possibility of capability re-use. This initial policy proved too restrictive, however, because it was found that sandboxes sometimes wanted to store copies of passed capabilities in global data-structures (for example when creating the compartmentalised version of `tcpdump` described in Chapter 5).

2. To resolve this difficulty the policy was changed such that the stack (and hence all capabilities derived from it) are marked as local, and only global, heap-allocated, capabilities can be passed between sandboxes. This prevents sandboxes from accidentally passing capabilities which could lead to vulnerabilities due to frequent stack re-use, but means that the only way to revoke a capability is through garbage collection or resetting entire sandboxes.

Note that the transition between these two policies did not require any changes to the hardware, illustrating the flexibility of the CHERI primitives. The exact treatment of memory allocation and re-use is a matter of software policy, and hence open to change in the future. Once a reliable model is established there may be opportunities for further optimisation in hardware, such as registers and instructions to support stack clearing as featured on PICA [77], but so far the existing CHERI primitives provide a simple and useful model.

3.2 CHERI Implementation

Jonathan Woodruff [24] implemented the CHERI ISA in a CPU written in Bluespec System Verilog [85–87]. Bluespec is a high-level synthesis language which allows rapid development of hardware designs which can be compiled to cycle accurate C++ simulation or Verilog for hardware synthesis. The implementation is a six-stage, in-order core which can be synthesised to run at 100 MHz on an Terasic DE4 board, with an Altera Stratix IV FPGA and 1GB of DDR2 RAM. It features 16 kiB, direct-mapped L1 data and instruction caches; a 64 kiB, 4-way associative shared L2 cache and a TLB with 144 entries, each of which maps two 4 kiB pages. The TLB consists of 128 entries stored in a direct-mapped table indexed by virtual address and a 16-entry fully-associative part which is used as a victim buffer for the direct-mapped table and to implement features of the MIPS ISA not compatible with the direct mapped TLB such as the immovable *wired* entries (used by the FreeBSD kernel to map the kernel stack). The CHERI extensions to the MIPS ISA are implemented as a separate co-processor which contains the capability register file and implements capability instructions and bounds checks.

Table 3.2: Summary of the features of the CHERI2 version used during benchmarking. Some parameters are configurable subject to timing and area constraints.

ISA	64-bit MIPS + CHERI extensions
Frequency	50 MHz on Altera Stratix IV FPGA
Pipeline	6-stage classic RISC, register forwarding
L1 caches	4-way (configurable), 16kB, write-through, 1 cycle latency
L2 cache	64kB, write-back, 3 cycle latency (L1 miss)
SDRAM	1GB DDR2@200 MHz, approx. 35 cycle latency (L2 miss)
TLB	64 entries, pairs of 4kB-256MB pages
Tag cache	4kB, 2-way associative (tags for 1MB memory)
HW Threads	Dual fine-grained multithreaded (configurable)

3.3 CHERI2

The original CHERI implementation was effective but had evolved over several iterations of the ISA and become unwieldy and hard to understand. There was also a focus on efficiency, sometimes at the expense of correctness, which made it difficult to modify for research purposes. Lastly, I found that aspects of the design were not ideal for experimentation, particularly the use of direct-mapped caches (for simplicity and increased clock frequency) which led to unpredictable effects during benchmarking.

CHERI2 is a second implementation of the CHERI ISA, written in Bluespec. It was started by Nirav Dave with the intention of creating a processor that could be formally verified via architectural extraction and automated theorem proving. It uses a single Bluespec *rule* (guarded atomic action) per pipeline stage and has a classic RISC microarchitecture with register forwarding paths to facilitate formal analysis. It is compatible with the original CHERI but does not share code except for supporting infrastructure such as the L2 cache, tagged memory controller, external memory bus and peripherals. CHERI2's main features are summarised in Table 3.2.

When beginning my research I decided to adopt CHERI2 because of its cleaner and simpler design, which made it easier to add the novel features I wanted to investigate, however I first had to bring it to feature parity with the original CHERI. I extended CHERI2 with all features necessary for it to boot the FreeBSD operating system, including much of the MIPS ISA, L1 caches and a memory management unit. I also added full support for the CHERI capability extensions and facilities to allow instruction level cycle-accurate tracing for debugging and performance analysis. Finally, I implemented fine-grained multithreading where the register files (both general purpose and capability) are replicated once per hardware thread and each thread has its own set of TLB entries and control registers such that threads can be treated by the OS as separate virtual processors. I also specified and implemented a programmable interrupt controller (PIC) which can be configured to direct external interrupts to given hardware threads and send interrupts between threads. These features allowed Brooks Davis to adapt FreeBSD's support for symmetric multiprocessing (SMP) to CHERI2, enabling me to evaluate the effect of hardware thread-level parallelism on domain crossing benchmarks, with relatively modest hardware and development-time overheads. Chapter 4 presents the results benchmarks running on single-threaded vs. multithreaded hardware.

Unlike the original CHERI, CHERI2 has 4-way associative L1 caches and a fully-

associative TLB. This configuration is better for running benchmarks because the associativity reduces the chances of encountering cache conflicts which can artificially skew the results, for example when running two versions of a function at different addresses one version might happen to conflict in the instruction cache with some other part of the benchmark, unfairly disadvantaging it. It is also more representative of real commercial processors which typically have at least two-way associativity in L1 caches.

A striking example of this was found during benchmarking on CHERI when I noticed that TLB misses were occurring in a simple microbenchmark with a small working set. After investigating using tracing I found that this is because forked processes share a large number of pages which differ only by their address space identifier, causing an unexpectedly high number of conflicts in the direct-mapped TLB. In one benchmark there were 16 conflicts out of a working set of 40 virtual pages – far more than expected and enough to overflow the large 16-entry victim buffer (2 entries in the buffer are reserved for fixed kernel pages). The problem is that forked processes necessarily share many TLB entries which differ only in their address space identifier (ASID) and CHERI's TLB used a very simple hash function which did not include the ASID, resulting in many conflicts. One solution would be to implement a more appropriate hashing function for the TLB, but I found that this caused compatibility issues with the MIPS ISA so on CHERI2 I chose to implement a fully-associative TLB.

Unfortunately, a disadvantage of associative structures is that they are relatively resource intensive to implement on an FPGA, especially given that direct-mapped structures can take advantage of fast block RAMs available on the Stratix IV. To mitigate this I implemented a two-level structure with 16 entry, fully-associative, single-cycle L1 TLB caches shared between the threads, backed by a multi-cycle TLB with 64-entries per thread. Even so CHERI2 currently runs at 50 MHz unlike the original CHERI's 100 MHz. This is because I prioritised correctness and usefulness as a benchmarking platform over absolute performance during implementation. I believe further optimisation of the implementation is possible, but so far this has not been an overriding goal.

3.4 Debugging, Tracing and Performance Counters

CHERI2 also has features designed to assist with debugging and benchmarking, specifically a debug unit, instruction tracing and user-space accessible performance counters.

The debug unit is separate from the main pipeline and accepts commands over a byte stream connected via JTAG. Commands are available for inspecting and updating registers, memory and capabilities and for controlling execution using pause, resume and breakpoints. These features are helpful for debugging but are also used for loading software images (e.g. kernels) in memory and controlling the embedded boot loader, which can be configured to wait for a particular register to be written before attempting to boot.

Additionally, the debug unit contains a circular trace buffer which logs an entry for the last 1000 completed instructions. Each entry contains the PC of the executed instruction, its encoding, a cycle counter and the result, such as any exception, register writes and the computed address for memory operations. This can be used to view a trace of the instructions leading up to a breakpoint or the processor can enter streaming mode, where the pipeline is paused when the buffer is full and resumed again when it is empty. I used this mode to obtain traces of micro-benchmark execution and analysed them (Section 4.3). The processor executes at full speed between trace batches and is paused whilst the trace

entries are downloaded, so the cycle numbers obtained from the traces are the same as those without tracing even though traces can only be downloaded at a fraction of full execution speed over the slow JTAG link.

Finally I added hardware counters to record a variety of events pertinent to performance analysis, namely elapsed cycles, committed instructions, TLB misses (data and instruction) and cache misses (L1 data and instruction). These counters can be read by software using the `rdhwr` (read hardware register) instruction, which is part of the MIPS instruction set. If enabled by the kernel these counters are directly readable by user-space programs, allowing them to be used for minimally disruptive measurement of benchmark performance (i.e. without resorting to costly system calls).

As well as being useful for debugging the combination of these features allowed me to analyse CHERI2's performance in benchmarks in detail, as shown in Chapters 4 and 5.

3.5 Multithreading on CHERI2

CHERI2 supports fine-grained multithreading, where instructions from multiple hardware threads can be in the pipeline simultaneously. This requires each control token to be tagged with a thread identifier and the register forwarding logic to be updated to only forward registers between instructions from the same thread, both relatively straightforward changes. The main cost of supporting multithreading is the additional storage required for all per-thread state, including general purpose, capability and control registers (such as those used by the kernel for manipulating the TLB). To minimise these costs I structured CHERI2 to make extensive use of block memory elements (BRAMs) available on the Stratix IV FPGA. For example, the architectural registers are stored in a BRAM which is indexed by register number and thread ID. This not only reduces the number of dedicated hardware registers used vs. an array of FPGA registers, but also eliminates logic because indexing into the BRAM replaces multiplexing between registers. However, not all per-thread state can be stored in this way because BRAMs have a one cycle delay between requesting an index and the data becoming available. To accommodate this I split the per-thread control state into two parts: a small part that is used early in the pipeline is stored in quickly accessible registers, and the remainder is placed in a BRAM which is fetched, processed and written back in the last three pipeline stages.

Multithreaded TLB

Another requirement to support multithreading is a separate set of TLB entries per thread. [2] This poses two problems: the additional storage requirements, and the need for a large content addressable memory (CAM). CAMs do not scale well at all, especially on an FPGAso I decided to split the TLB into two levels. The first level TLBs are 16-entry fully-associative caches that are shared between threads, and accessed in parallel with the

[2] Although it is interesting to consider a processor with a single logical TLB shared between threads this would not be so useful for supporting commodity operating systems such as FreeBSD, as they are written to run on processors with multiple independent cores or heavyweight threads such as Intel's Hyperthreads. There would also be difficulty resolving this with the software managed TLB misses of the MIPS ISA: races could occur if another thread were to take a TLB miss whilst the first is still being handled.

L1 cache lookups in the instruction and data fetch stages. The second level has 64-entries per thread and implements the ISA visible TLB as described in the MIPS specification.

Load linked/store conditional support

MIPS provides the load linked and store conditional (`ll`/`sc`) instructions for atomic memory operations in user space code. This is a form of optimistic concurrency which is essentially a hardware transactional memory for operating on a single word at a time. The load linked is an ordinary load which begins a transaction and the store conditional completes it by writing the data only if there was no conflict and returning a value indicating success or failure.

On a single-threaded processor these are easily implemented by recording the address of the last load linked operation and invalidating it if an exception (e.g. a timer interrupt) occurs or there is a non-conditional store operation to the address. The store conditional then checks against the recorded address to see if the operation should succeed.

On a multithreaded processor this is more complicated because each thread must have its own load linked address slot, and a write by any other thread could potentially invalidate it, causing the store conditional to fail. One way to implement this would be to store one load linked address per thread in a content addressable memory which is consulted and updated on every store by any thread.

Initially I thought that I could increase scalability by making an addition to the level one data cache tag to record the ID of the last thread to load link a given address. The store conditional instruction then checks in the cache to confirm that the thread was the last one to load link the address before proceeding. Unfortunately this scheme can lead to live lock as two threads competing to perform an atomic operation can repeatedly 'steal' the cache line by performing the load link without ever successfully completing a store conditional. Software must also be careful not to invalidate the cache line between the `ll` and `sc` which effectively means that no memory accesses can be performed in the interim (the MIPS specification explicitly says this will result in undefined behaviour).

In practice I found these restrictions were too onerous to meet in software, so I reverted to the simple per-thread register which scaled acceptably and did not form part of the critical path.

Thread scheduling

An important factor of multithreaded processors is the scheduling of threads into processor cycles. For simplicity I adopted a round-robin scheme where the IDs of all running threads are placed in a queue from which the head is dequeued on each instruction fetch and returned to the tail of the queue. When an instruction executes the `WAIT` instruction it is removed from the queue and resumed when it receives an interrupt, meaning that idle threads do not consume CPU cycles polling.

Overall adding multithreading support to CHERI2 required a modest number of code changes and was affordable in terms of hardware resources. It allowed me to evaluate the effect of hardware parallelism on compartmentalisation without the complexity of implementing a multicore processor capable of fitting on the FPGA. The results of this are presented in Section 4.5.

3.6 Hardware overheads of CHERI support

Hardware support for capabilities comes at some cost in terms of silicon area and, potentially, clock frequency. The additional hardware required is:

1. The capability register file, containing 32 capabilities, each 257 bits (256 bits plus one tag bit) and a separate program counter capability register
2. The capability co-processor implementing the capability manipulation instructions
3. Bounds checking logic in the instruction fetch and memory access stages to compare fetched addresses against capability bounds
4. One additional bit per 256-bit cache line in the L1 and L2 caches to store the capability tags
5. A tag cache/controller which emulates tagged memory by storing tag bits in a dedicated portion of SDRAM

The first two of these are the most significant costs in terms of area and frequency, owing to the wide data paths associated with capabilities. CHERI2 supports conditional compilation of the capability support so I compared the FPGA resource utilisation as reported by Quartus with and without CHERI extensions and found that there was an approximately 20% increase in ALMs (Adaptive Logic Modules [88]) and 5% in block memory bits for the CPU core and memory hierarchy (this excludes peripherals such as the soft SDRAM controller and the system bus). The estimated maximum clock frequency reported by the timing analyser dropped from 55MHz to 53MHz owing to a lengthening of the critical path, which was in resolving the branch condition in the execute stage and computing the next PC to fetch (CHERI includes a 'branch if tag set' instruction to allow efficient compilation of null-pointer checks). Note that I did not focus on optimising the design for area or frequency because I was more concerned with correctness, so these numbers are only a rough indication of the overhead that might occur in a more optimised design.

The space overhead of tagged memory is small as only approximately 0.4% area/memory overhead is required to store one extra tag bit per 256-bit location. A naive tag controller could result in a doubling of the required memory bandwidth because an extra memory access would be required for every read or write in order to fetch the tag data separately, however a small tag cache significantly reduces this because a single 32-byte memory operation can read/write the tags for 256 32-byte cache lines. Furthermore, tags only need to be accessed in memory if there is both an L2 and tag cache miss. An alternative to the separate tag controller would be an SDRAM controller which directly supports tag bits in a manner similar to the ECC (error-correcting code) bits on some modern systems.

Overall I think the additional costs of capabilities are acceptable considering the many benefits of memory safety and compartmentalisation. I would expect a commercial deployment to be more optimised in terms of the implementation and elements of the ISA, which is currently designed to facilitate research and design space exploration at some cost to efficiency (viz. the unused reserved space in the capability format). In particular, compressed 128-bit capabilities would reduce the cache footprint and memory bandwidth

costs associated with the larger pointer size, and could potentially save some silicon area owing to smaller data paths, but might increase the amount of logic used for capability manipulation and bounds checking, depending on the representation used.

3.7 Limitations of CHERI2 as a research platform

I chose to target an FPGA implementation because I lacked the manpower and resources required for a custom ASIC and believe FPGAs can provide more realistic experimental results than software simulators for two reasons. Firstly, because FPGAs are subject to similar timing and resource constraints to ASICs anything implemented on FPGA necessarily has a feasible silicon implementation, although the details and trade-offs may differ. This does not apply to simulation, where careful consideration is needed to ensure that any simulated architecture could be realistically implemented in silicon. Secondly, because FPGAs run significantly faster than simulators it is possible to run much more extensive, and hence realistic, benchmarks. In Chapter 5 I use this to perform long-running macro-benchmarks under a full commodity OS, a feat which would be much more difficult on a slow running simulator.

That said, CHERI2 is not an ideal model for a real commercial processor in several respects: due to development time and FPGA resource constraints it has a relatively simple pipeline compared to a modern out-of-order, superscalar processor and also a small L2 and very fast SDRAM in relation to the core frequency. I consider these acceptable limitations because during benchmarking I use CHERI2 itself as a baseline and in some experiments (e.g. Sections 4.3 and 4.5) I observe orders of magnitude differences in relevant variables, therefore the results are unlikely to be so wildly affected by micro-architectural choices as to invalidate the conclusion that domain crossing using `CCall` is much more efficient than IPC-based methods. In other experiments, such as Section 4.6, I analyse the current implementation and make incremental improvements, the details of which may differ on other implementations but the principles of which are still likely to be applicable. To reinforce this I ran the same benchmarks on the original CHERI implementation, which has a rather different micro-architecture, and obtained very similar results. Finally some metrics I use, such as instruction count, cache footprint and system call count are independent of the micro-architecture, being more dependent on the ISA and benchmarks themselves, both of which I chose to be representative of real processors and applications (MIPS being fairly typical of RISC architectures).

To obtain an absolute measure of performance I ran the popular Dhrystone 2.1 integer benchmark [89] compiled with `gcc` 4.2.1 at the default optimisation level (-O). Although Dhrystone is not an especially realistic benchmark owing to its small size and behaviour unrepresentative of real applications I chose it because of its ease of compilation and the wide availability of reported results. CHERI2 executes around 43.3×10^3 Dhrystone iterations per second, or 0.49 DMIPS/MHz[3]. For comparison the MIPS Technologies 5Kc, a single-issue, 6-stage, synthesisable MIPS64 processor, performs about 92.9×10^3 Dhrystones per second running at 40MHz, or 1.32 DMIPS/MHz [90]. This indicates that CHERI2 is not as efficient as a comparable commercial processor, which is unsurprising given the research timescale and resources available. Based on observing traces I conclude

[3]DMIPS means Dhrystone interactions per second normalised to the VAX 11/780 which achieved 1757 Dhrystones per second. This can be further normalised by the clock frequency to give DMIPS/MHz.

that the gap in performance is due to cycles per instruction: CHERI2's CPI in the benchmark was 1.92 owing to a significant proportion of memory operations taking more than one cycle and a single divide instruction taking 66. Nevertheless I believe experiments using CHERI2 can be used to draw conclusions about the CHERI ISA features as they might be implemented in a heavily optimised commercial processor, providing the results are significant enough and consideration is given as to how they may be affected by alternative micro-architectures.

3.8 FreeBSD and CHERI

Robert Watson, Brooks Davis and others ported the FreeBSD operating system to the CHERI prototype, creating a variant called CheriBSD. Few changes to the existing MIPS port were required to support the capability features (in fact unmodified kernels will run successfully providing applications do not use the capability extensions). The minimal changes necessary to support CHERI include:

1. Additions to the context save/restore code to preserve the capability register file on entering and leaving the kernel.

2. Handlers for the new capability exceptions generated when applications violate capability restrictions (for example by performing an out of bounds memory access). The default action is to print debugging information and terminate the process, although it is also possible to configure a signal handler to deal with exception, such as by resetting the CHERI object responsible.

3. Routines to correctly save and restore tag bits when swapping memory to disk.

On top of this CheriBSD implements some features to enable sandboxing of applications using the CHERI ISA. This requires handlers for the traps generated by the `CCall` and `CReturn` instructions which implement the additional checks and semantics required for secure transitions between domains, as detailed in Table 3.1. CheriBSD also adds a check in the system call path to allow sandboxes to be prevented from invoking system calls directly. This check consults the program counter capability (`$pcc`) of the caller to see if it has the SYSCALL permission and prohibits the system call if it is not set. This means that sandboxed code may only make system calls indirectly via a *system proxy* class invoked via `CCall`, allowing the application to perform extra checks according to its own policy before permitting the system call. This affords great flexibility without embedding a large MAC framework into the kernel, but at the cost of additional domain transitions on each system invocation. This price is acceptable because many sandboxes perform pure computation and do not need to invoke system calls, whilst the cost of `CCall` transitions is modest for those that do, as shown by the benchmarks in Chapter 4.

3.8.1 Sandbox implementation

CheriBSD implements sandbox objects which consist of the traditional three segments: code, data and stack referred to by `$pcc`, `$ddc` and `$scc` respectively. Processes begin with *ambient authority*, an entirely backwards compatible environment where all privilege is granted, and can load sandbox *classes* using a special loader which reads statically linked

ELF (Executable Linkable Format) files. Once loaded, *instances* of the class can be created which each have their own stack and data sections. The data section is initialised from the global data section of the sandbox ELF file and also contains space for a small local heap.

Sandbox instances can then be *invoked* using the `cheri_invoke` function which is part of the `cherilib` library. This function prepares for and performs the `CCall`, entering the sandbox via its `invoke` function, which sets up the sandbox environment and branches to appropriate method based on the `CCall` arguments.

Inside the sandbox memory access is restricted to the sandboxes memory segments and system calls are prohibited according to the `User_Syscall` permission bit in `$pcc`. To permit limited access to system resources a specialised `libc` implementation in the sandbox proxies system calls via the *system object*, which is a more trusted sandbox object which executes with the `User_Syscall` permission. This design minimises exposure of the large system call ABI to untrusted sandbox code.

3.8.2 Library compartmentalisation

The sandbox `libc` and system object is a special example of *library compartmentalisation*, where an existing C-language API has its functionality moved into a sandbox transparently using a thin, backwards-compatible wrapper which translates between the legacy API and a sandboxed version.

Library compartmentalisation is a very powerful strategy which allows all code using a compartmentalised library to gain the benefits of compartmentalisation without requiring any code changes in the applications. CHERI supports library compartmentalisation well because it allows pointers in the library API to be passed using capabilities, avoiding the need to copy data into and out of the sandbox. Section 5.3 demonstrates this using the example of `zlib`.

3.9 clang/LLVM and CHERI

David Chisnall extended the MIPS port of the LLVM compiler framework and clang C-compiler with features to support the CHERI ISA. The compiler can function in either hybrid mode, for greater backwards compatibility, or pure capability mode, for maximum use of CHERI features.

In hybrid mode pointer types can be annotated in the source code with a `__capability` modifier which causes the compiler to emit capability instructions for manipulating and dereferencing those pointers, instead of standard MIPS instructions. The application binary interface (ABI) for function calls is extended to accommodate passing capability arguments via capabilities registers, and the compiler attempts to determine appropriate bounds when capability pointers are initialised, for example when taking the address of an object whose size is known at compile time, or when casting the pointer returned by a call to `malloc` . These features allow existing C programs to gradually adopt CHERI features and gain the memory safety benefits without being faced with major compatibility problems.

In pure-capability mode (also called sandbox mode), all pointers are implemented with capabilities including the stack pointer and (optionally) function return addresses. This

provides a very high level of memory safety which includes integrity guarantees capable of thwarting powerful attack techniques such as return-oriented programming.

Finally, clang implements support for `CCall` via a special ABI and calling semantics. If a function is annotated with the `__cheri_ccall` attribute clang uses a different calling convention: the code and data capabilities for the invoked object are passed in `$c3` and `$c4` and a method number is passed in `$v0` (which normally holds the function return value). This convention avoids using existing argument registers for the extra method invocation parameters as this would reduce the number of arguments registers available and would also require extensive shuffling of arguments when a ordinary function call is used to wrap a capability invoke with identical arguments (as might occur in a backwards compatible interface to a compartmentalised library). The attribute also causes the compiler to emit code to clear all unused argument registers so as to avoid leaking capabilities or data to the callee. Rather than using the `CCall` instruction directly the compiler emits a call to the `cheri_invoke` library function. This routine is shared by all invoke points and performs actions that do not need to occur in the trusted kernel context, specifically saving all callee-save state and clearing non-argument registers.

In the receiving sandbox an invoke handler function uses the passed method number to select the target function, which must be tagged with `__cheri_ccallee` so that the compiler can clear any unused return value registers on returning. The return path is the reverse of the calling path, but is slightly less costly because fewer checks are needed on the argument registers.

CHAPTER 4

CHERI Bench: domain crossing microbenchmark

This chapter contains an analysis of security domain crossing under FreeBSD, comparing traditional techniques with the CHERI extensions. I created a microbenchmark that performs transitions between security domains using different isolation mechanisms and used this to measure the performance cost. Specifically, I measured the overhead of supporting the CHERI extensions in the FreeBSD kernel, compared the various domain crossing techniques with a method invocation using a CHERI `CCall`, and made a detailed analysis of the costs of a CHERI invoke.

4.1 The Benchmark: `cheri_bench`

I wrote a C program, called `cheri_bench`, that performs domain transitions into a sandbox using seven different mechanisms. In each case the sandbox simply copies data from an input to an output buffer, thus measuring the cost of entering and leaving the sandbox as well as transferring data in and out, including cache and TLB overheads.

The mechanisms were as follows:

func A normal call to a function that performs a `memcpy` and returns. This provides no isolation, but gives a baseline against which to compare other mechanisms.

invoke A method invocation on a `libcheri` object that copies data between two buffers passed via capabilities. `libcheri` uses `CCall` to transition between protection domains; the compiler also generates code to clear unused argument and return-value registers.

pipe A forked sandboxed process connected via a UNIX pipe. Data is sent and echoed back again via the pipe. This IPC model is used in most privilege-separated applications, such as `OpenSSH`.

socket Similar to the **pipe** case but using a UNIX socket instead of a pipe. Pipes perform better than sockets on FreeBSD due to optimisations using virtual memory to avoid copying data.

shmem+pipe A forked process communicated with via a pipe and shared memory. The pipe is used to synchronise by sending a length argument, while the `memcpy` occurs via shared memory. This IPC model is used in larger-scale compartmentalised applications (such as between Chromium's browser process and renderer sandboxes) where bulk data is shared by compartments.

shmem+sem Processes using shared memory can use POSIX semaphores for communication rather than pipes. This is potentially more efficient as the semaphore can be implemented entirely in user space using atomic memory operations. If multiple hardware threads are available (i.e. on multicore or multithreaded CPU) then this can be achieved without a context switch, although achieving this depends on careful scheduling of threads and requires some polling.

pthread+sem This is similar to **shmem+sem** except using POSIX threads instead of a forked process. Threads share an address space and therefore do not provide security isolation, however this case provides an interesting way to assess the cost of shared memory using virtual memory.

`cheri_bench` uses a cycle counter (accessed via the user space `rdhwr` instruction) to measure the number of cycles per iteration for each benchmark. I also augmented the CHERI processor with hardware TLB miss counters to provide a low overhead method of accurately observing the TLB behaviour. An iteration comprises entering the sandbox, copying data from input to output, and returning to the caller. The caller reads the output buffer between each iteration .

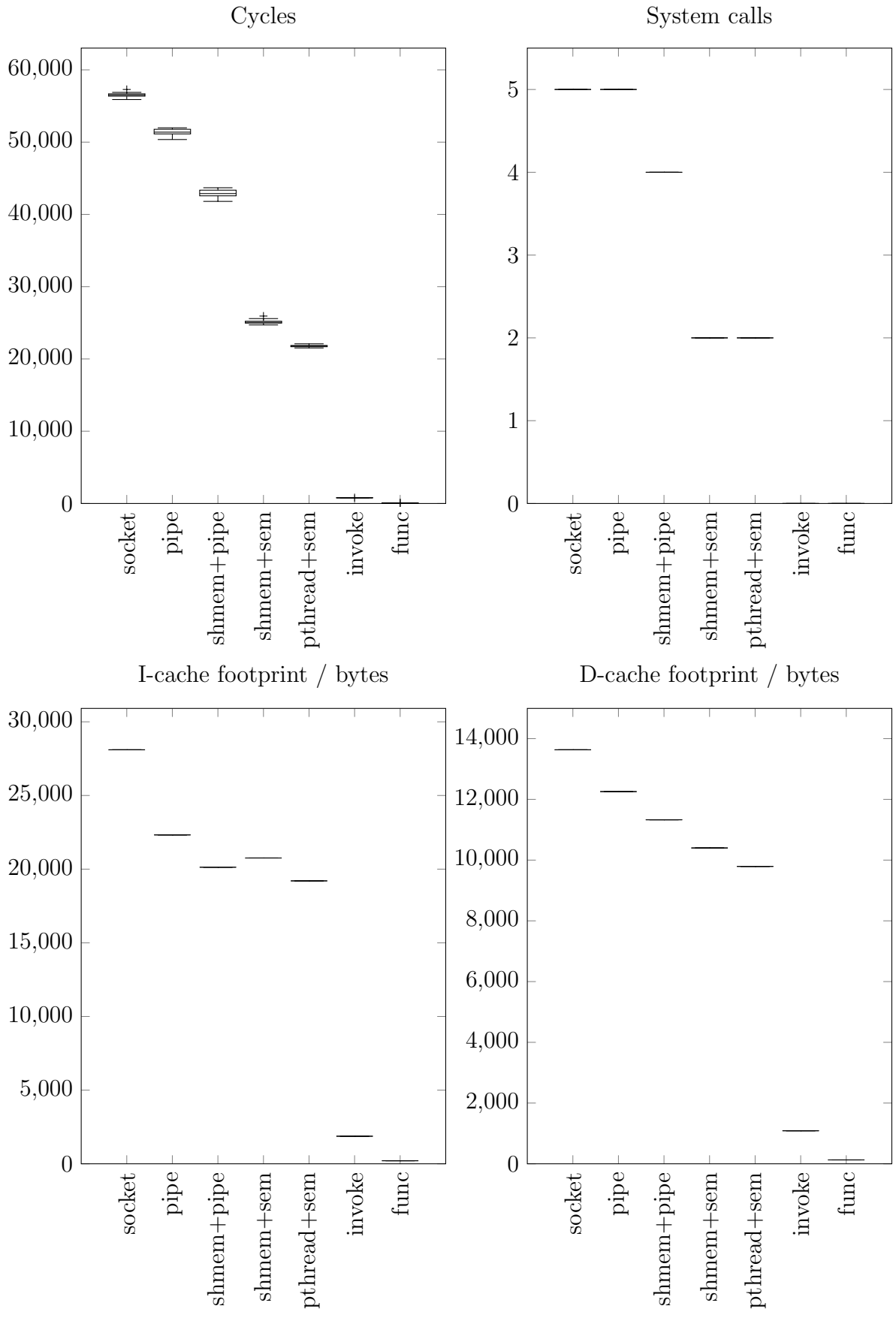
4.2 Experimental setup

The experiments in this chapter were run on CHERI2 as described in Section 3.3, synthesised to run on a Terasic DE4 FPGA development board at 50 MHz. I used a quiesced version of CheriBSD with a reduced set of peripherals (console UART only) and a memory root filesystem to minimise noise due to interrupts and I/O. For detailed analysis of short experiments (such as those in Section 4.3) I used CHERI2's cycle-accurate instruction tracing features, but for longer running ones I used the user-space performance counters because tracing longer running benchmarks can take a long time.

4.3 Experiment 0: Benchmark characteristics

I characterised the `cheri_bench` benchmark by tracing 16 iterations of each domain crossing method using CHERI2's cycle-accurate tracing, then analysed the traces to extract information about the runtime behaviour. I used a payload of 32 bytes since this is the smallest payload which does not cause degenerate behaviour (this is the size of the word copied by `memcpy` using the capability registers). I discarded the three initial iterations of each benchmark to ensure the caches and TLB were warm. Figure 4.1 shows the resulting boxplots for four metrics: cycles per iteration (round trip), number of system calls per iteration, instruction cache footprint (#PCs $\times 4$ bytes) and data cache footprint (#lines touched $\times 32$ bytes). The benchmarks are very deterministic so there is no variation between iterations for any metric except cycles, where there is some due to the random

Figure 4.1: Benchmark characteristics per iteration



cache replacement policy. The benchmarks are short enough that I was able to obtain 16 consecutive iterations during which no timer interrupts fired.

Note that there is a close correlation between the number of system calls used by each domain crossing method and the number of cycles taken, reinforcing that system calls are an expensive privilege escalation mechanism. Examination of the trace reveals that this is not due to the hardware cost of the system call (which is about 10-15 cycles per `syscall`), but because of the kernel overhead of handling the call, namely saving the thread context, setting up the kernel's runtime state (stack etc.), locating the appropriate handler, copying arguments to and from kernel buffers and invoking the scheduler to resume the appropriate thread. The **socket** and **pipe** methods are the most expensive and require 5 system calls as follows:

1. A `writev` in the caller to write the payload length and data to the socket
2. A `read` in the sandbox to receive the length
3. Another `read` in the sandbox to receive the data (these cannot be combined into a single call as for the write because the length of the data is not known in advance)
4. A `write` in the sandbox to echo the data back
5. A `read` in the caller to receive the returned data

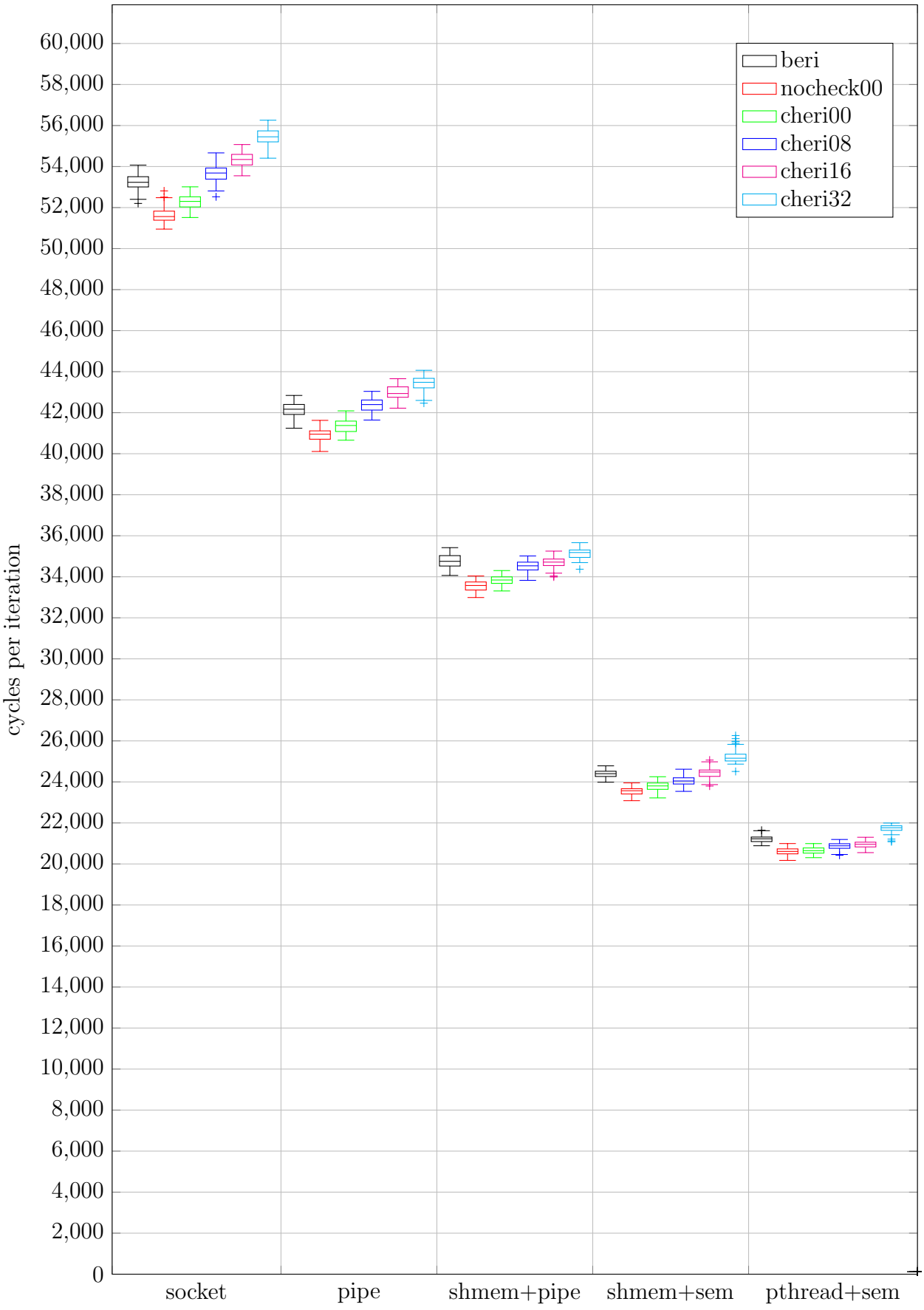
The **shmem+pipe** case takes one fewer system call since the data is copied via shared memory instead of being read from the socket, but still requires four system calls to read and write the length argument and for synchronisation on return. The semaphore based cases only use one system call in each direction to suspend the current thread when waiting on the semaphore, but their I-cache footprint is still around 20 kiB, exceeding the size of the 16 kiB L1 cache.

The CHERI **invoke**, on the other hand, uses no system calls. Although it does enter the kernel via the `CCall` and `CReturn` traps, these are handled by dedicated kernel handlers, avoiding much of the overhead of system calls due to their synchronous, single-purpose nature. As a result **invoke** has cycle costs and cache footprints an order of magnitude lower than the traditional domain crossing methods. However, **invoke** still has significant overhead relative to an unprotected function call. These costs are investigated further in Section 4.6.

4.4 Experiment 1: BERI vs. CHERI kernels

Before further analysing the cost of domain crossing using the CHERI extensions I wanted to establish how they affected the performance of the system when they are not being used. The FreeBSD kernel must save and restore the capability register file on every context switch. This is potentially a significant cost given the size of the CHERI capability register file (32×256 bits = 1 kiB), in particular the impact this could have on the L1 data cache. To test this I compiled a version of `cheri_bench` that only uses the traditional isolation techniques (**socket**, **pipe**, **shmem+pipe**, **shmem+sem** and **pthread+sem**) . Since this benchmark does not use capabilities it is safe to partially disable the kernel's context save/restore routine to estimate the costs for smaller capability register files. Note that the backwards compatible design of the CHERI ISA means that capability oblivious code,

Figure 4.2: CHERI kernel overhead



even the kernel, can run on the same CHERI processor without modification (with the obvious caveat that disabling the capability save/restore removes the isolation between processes, meaning this is only useful for benchmarking purposes).

Figure 4.2 gives box plots for the cycles per round trip domain transition for each IPC method and a variety of different kernels. The 'beri' kernel is FreeBSD compiled for standard MIPS64, whilst cheri00 – cheri32 target CHERI with different numbers of capability registers being saved. The 'nocheck00' kernel is the same as cheri00 except that it also has the system call authorisation check on `$pcc` disabled (see Section 3.8), meaning that it has no kernel overhead as a result of supporting the CHERI extensions. Note that this version actually outperforms the beri kernel because the CHERI kernels use a modified `memcpy` routine that copies data via the capability registers instead of one double word at a time. This is necessary to preserve the capability tags, but also results in a more efficient memory copy since each loop iteration transfers 32 bytes instead of 8. Therefore the fairest assessment of kernel overhead is obtained by comparing nocheck00 with cheri32. Within the CHERI kernels the costs increase as the size of the capability register file grows, as expected. In the worst case (**socket**) there is a difference of 3900 cycles between the median of the nocheck00 and cheri32 kernels, about 780 cycles per system call, or 7.5%.

This experiment shows that the cost of saving the CHERI context can be significant but that there is scope for reducing this cost by limiting the size and number of the capability registers. Since CHERI was designed as a research platform the configuration of 32, 256-bit capability registers was chosen to maximise flexibility, however this may represent an unrealistically large context. Ongoing research indicates that a smaller number of capability registers would suffice and that the in-memory capability representation can be compressed to further reduce memory footprint. Also, other large registers sets exist in modern processors, for example recent AVX-512 vector extensions now shipping in Intel x86 processors have 32, 512-bit registers. With this in mind I believe that CHERI has acceptable kernel overhead, especially given the memory safety benefits and potential accelerated domain crossing as shown in the next experiment. .

4.5 Experiment 2: Comparison of domain crossing mechanisms

I performed an experiment to compare the cost of the different sandboxing mechanisms and explore the way they scale with the amount of data transferred across the security boundary. I varied the payload from 32 bytes to eight megabytes in powers of two and computed a trimmed mean after discarding outliers caused by start-up costs and timer interrupts.

Figure 4.3a shows the absolute cycle costs for each method and Figure 4.3b shows the same data after subtracting the function call baseline. Error bars show the sample error in the absolute case, and the square root of the sum of the variances of the sample and base cases for the overhead graph . We see that all methods scale approximately linearly with the size of the `memcpy` but have different fixed costs due to the domain crossing/synchronisation mechanism used. The `CCall` has a minimum overhead of around 700 cycles and remains low, whilst the other cases have much higher overhead (20,000 cycles or more) due to the OS system calls used for synchronisation. Slight changes of

gradient, most visible in the **func** case, indicate cache and TLB size boundaries. The limit occurs when the data exceeds the capacity of both caches and the TLB which, due to the linear access pattern, means every access is a miss even on repeated invocations. Note that the **pipe** and **socket** cases perform extra data copies and so remain five to ten times more expensive for the larger data sizes, whilst all other cases have fixed overhead and converge with the **func** baseline when the fixed costs are amortised against the large `memcpy` costs.

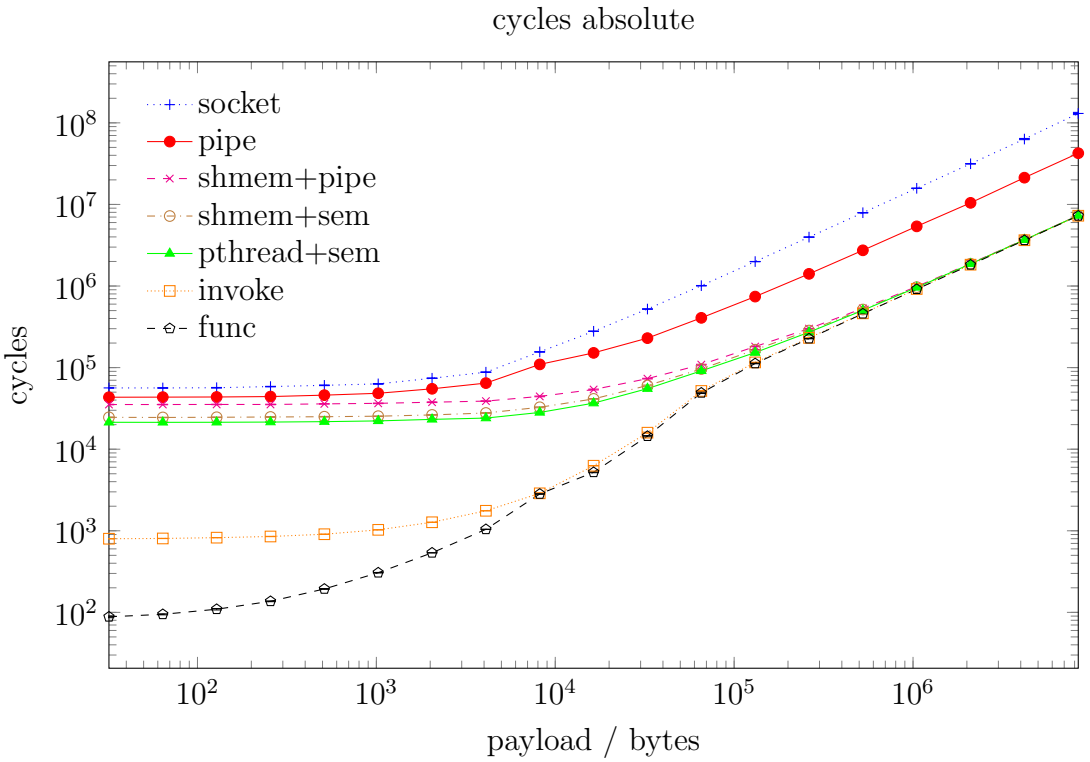
The overhead graph (Figure 4.3b) shows this in more detail. The **pipe** and **socket** rapidly leave the top of the graph due to linear overhead. The other cases have some initial fixed overhead which increases slightly as the extra code and data used for domain crossing begin to conflict with the payload. For **pthread+sem** this peaks at 32kB (when the source and destination arrays completely fill the 64kB L2 cache), but for the **shmem+pipe** and **shmem+sem** cases the overhead continues to grow due to conflicts in the TLB – twice as many TLB entries are required to share memory between two processes as to map it in a single address space. This can also be seen in Figure 4.4, where the process based isolation causes TLB misses with half the payload size of the within-address-space methods.

To evaluate the effect of multithreading on the benchmark I ran two versions of the experiment. In the first version already discussed both parent and any child processes were pinned to a single thread using FreeBSD's `cpuset_setaffinity` function, effectively treating the hardware as single threaded; in the second version, the parent processes was pinned to hardware thread 0 and the children to thread 1 ensuring use of multiple hardware threads. Figures 4.5 and 4.6 show the results for the dual threaded case which resemble the original graphs except for some notable differences. Figure 4.7 shows the relative performance of the multithreaded version over the single thread version. The **func** and **invoke** cases are inherently single-threaded so remain roughly the same , but in general the multithreaded cases are *slower* because the OS must perform an inter-thread interrupt to resume the waiting hardware thread, incurring extra fixed overhead. This is true because the benchmark is essentially synchronous meaning that there is no parallelism for the multithreaded hardware to exploit. The significant exception to this is the cases using semaphore based synchronisation, where the overhead for small payloads is greatly reduced to a minimum of around 1000 cycles. This is because synchronisation in this case is achieved simply by writing into memory and polling for the update in the receiving thread, so the communication can occur with very low latency and without any system calls or interrupts. Polling is inefficient, however, as the polling thread consumes resources without achieving useful work. This explains why the overhead increases as the payload increases – the polling thread slows down the thread performing the `memcpy`, making it about 50 percent slower and meaning that the performance saving is quickly eroded until the relative performance plateaus at about two times slower than the single threaded case. FreeBSD's semaphore implementation features adaptive mutexes which attempt to mitigate this problem by imposing a limit on the length of polling after which the thread invokes the OS scheduler to put the thread to sleep[1]. This is why the relative overhead decreases after a point when the semaphore cases effectively revert to their

[1] I obtained unexpected results during initial runs of this experiment until I discovered that it takes up to 50 iterations for the adaptive mutex to discover the optimal behaviour. The results shown use an increased number of iterations with the first 50 discarded. This emphasises the difficulty and unpredictability of using asynchronous mechanisms for compartmentalisation.

Figure 4.3: Comparison of domain crossing methods – cycle costs, single thread

(a) Absolute cycle cost (log-log)



(b) Cycle overhead vs. function call (log-linear)

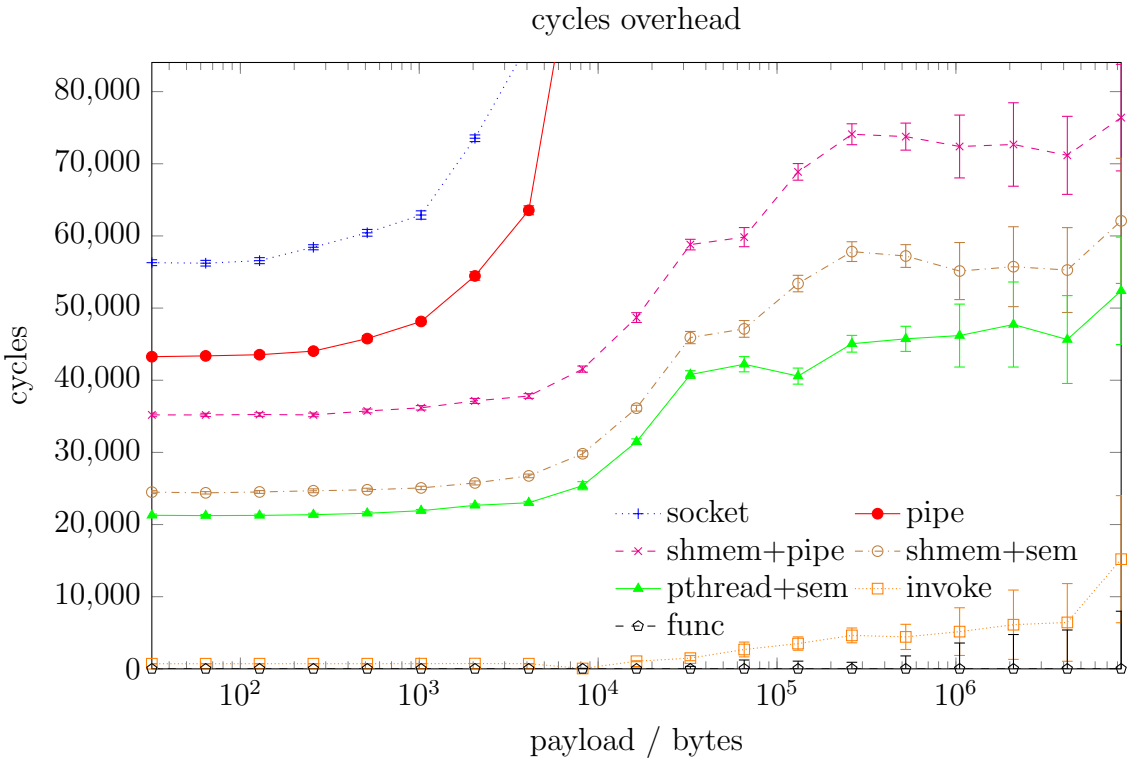
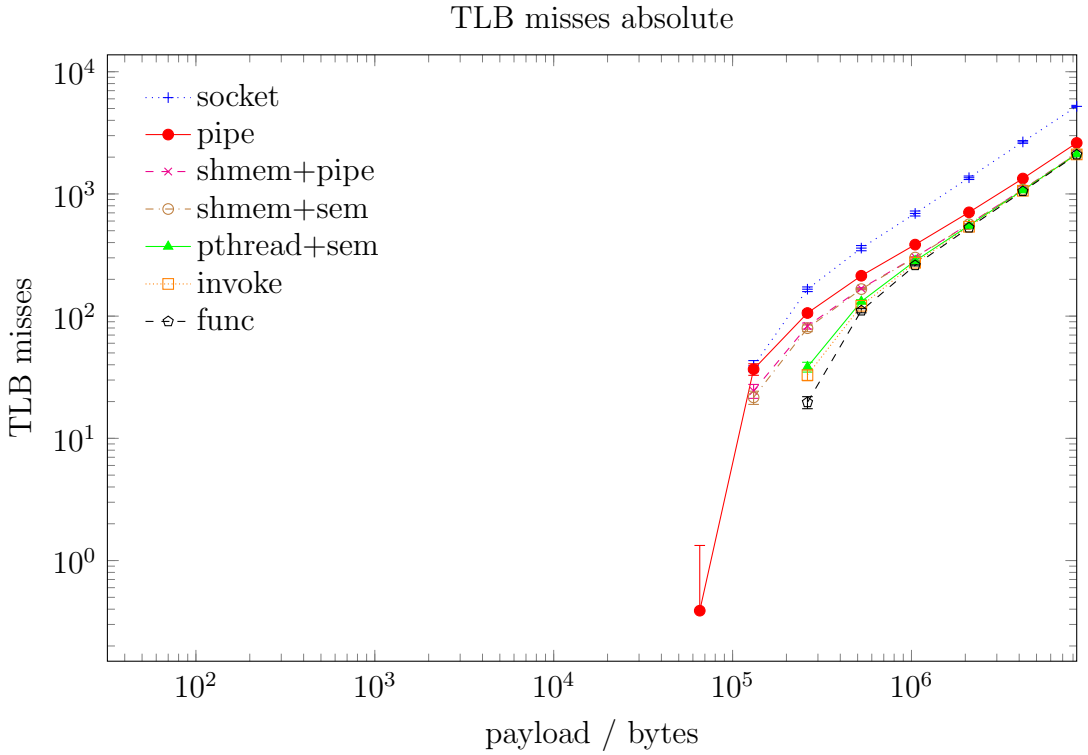


Figure 4.4: Comparison of domain crossing methods – TLB costs, single thread

(a) Absolute TLB misses (log-log)



(b) TLB miss overhead vs. function call (log-linear)

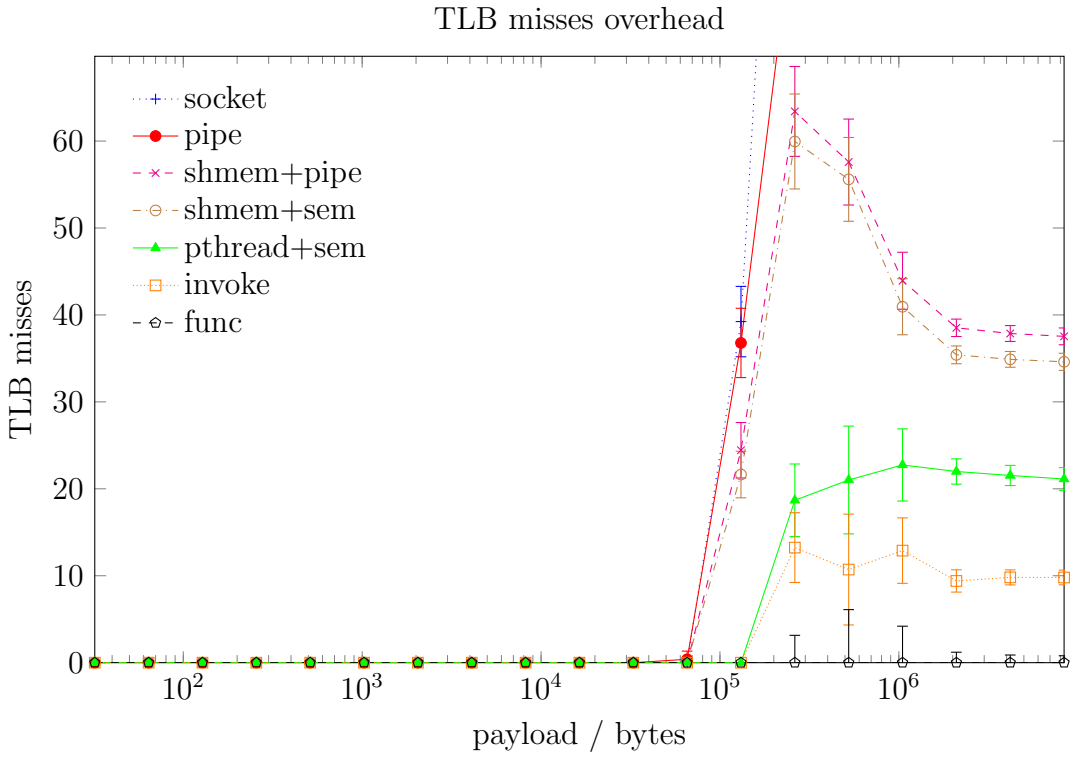
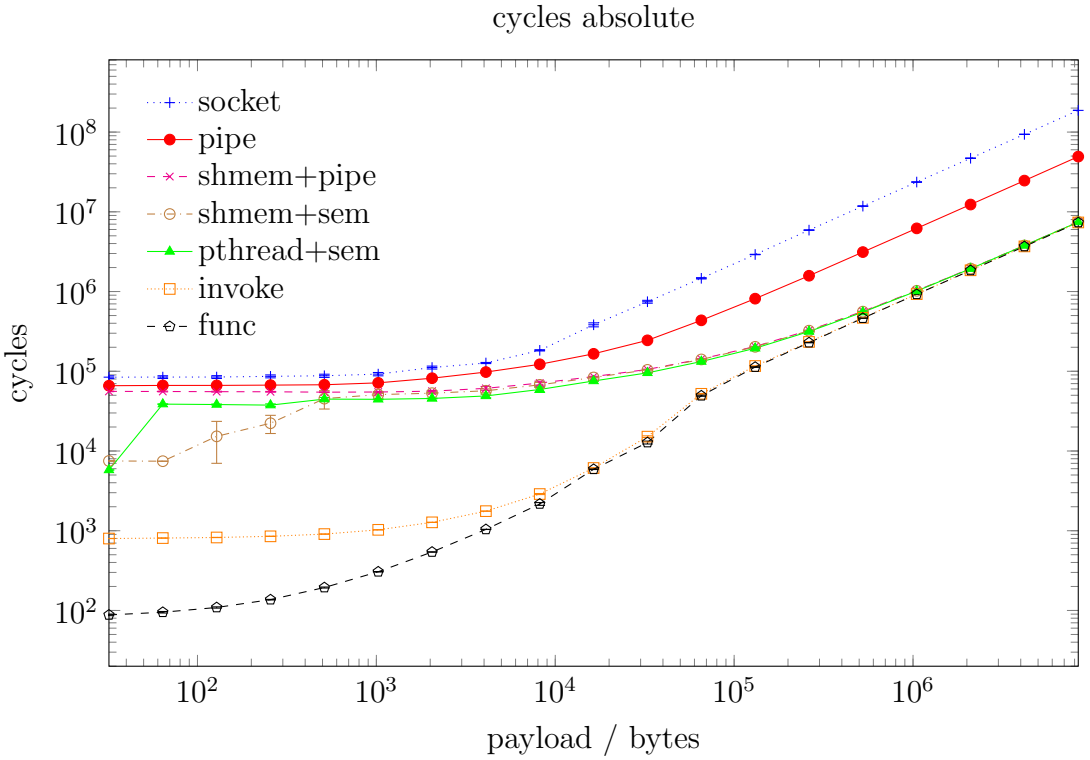


Figure 4.5: Comparison of domain crossing methods – cycle costs, dual thread

(a) Absolute cycle cost (log-log)



(b) Cycle overhead vs. function call (log-linear)

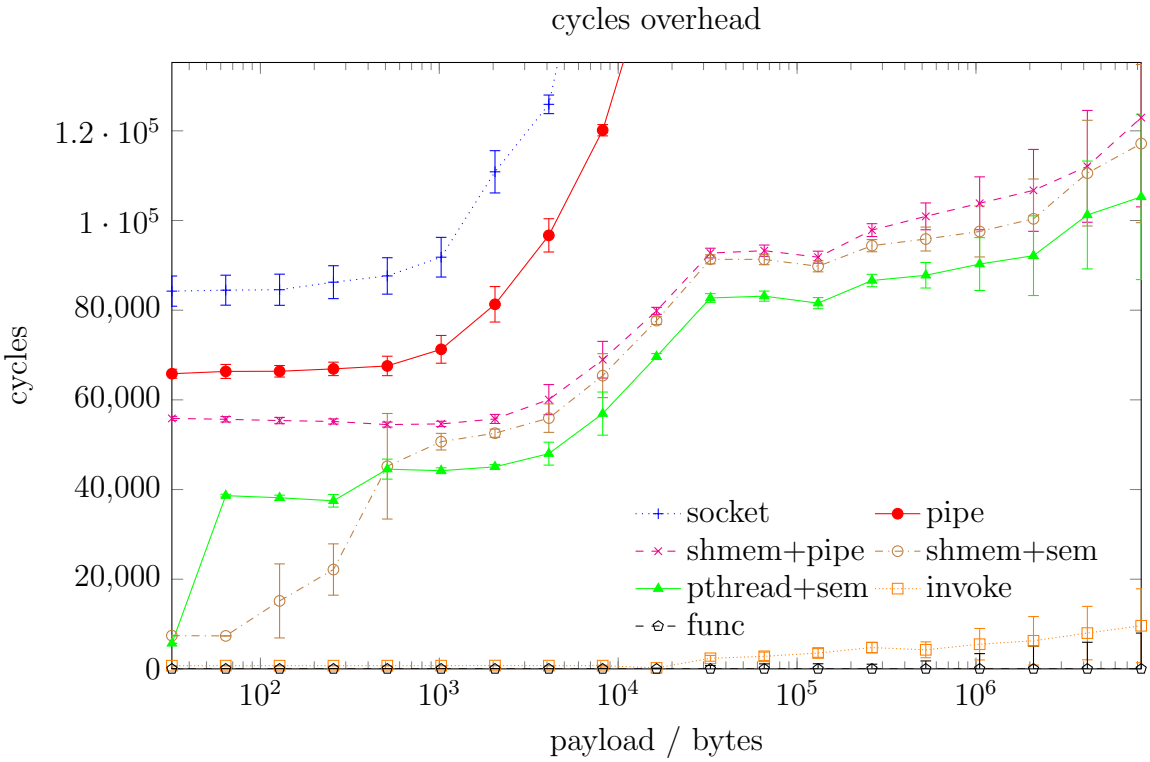
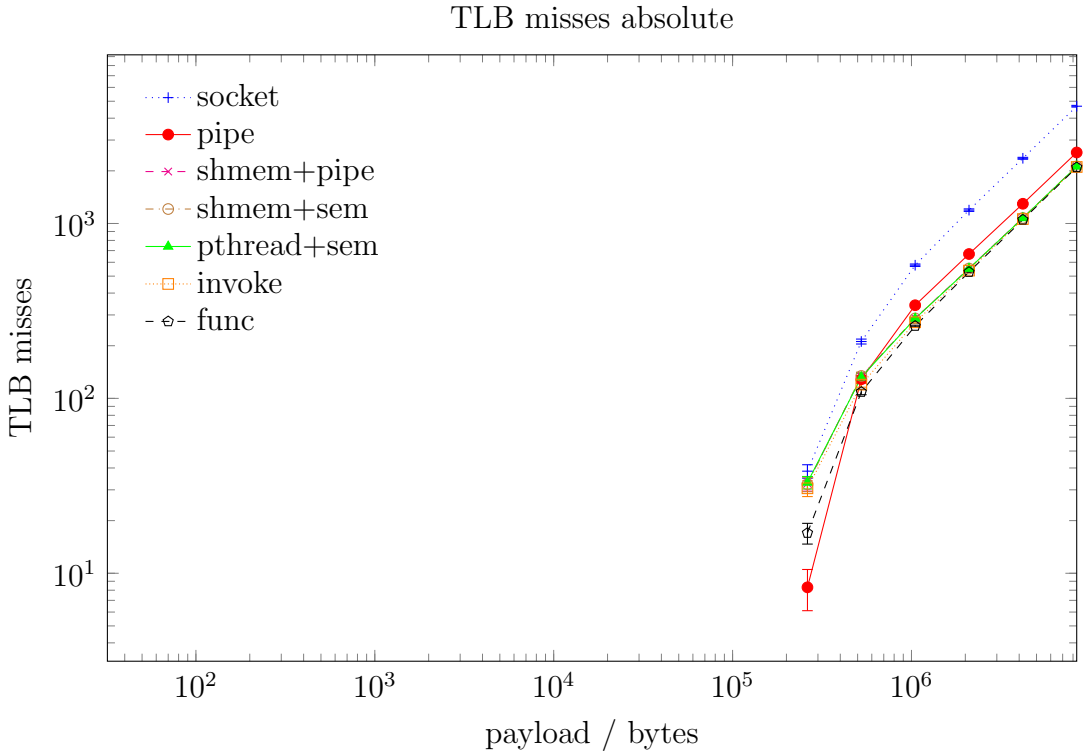


Figure 4.6: Comparison of domain crossing methods – TLB costs, dual thread

(a) Absolute TLB misses (log-log)



(b) TLB miss overhead vs. function call (log-linear)

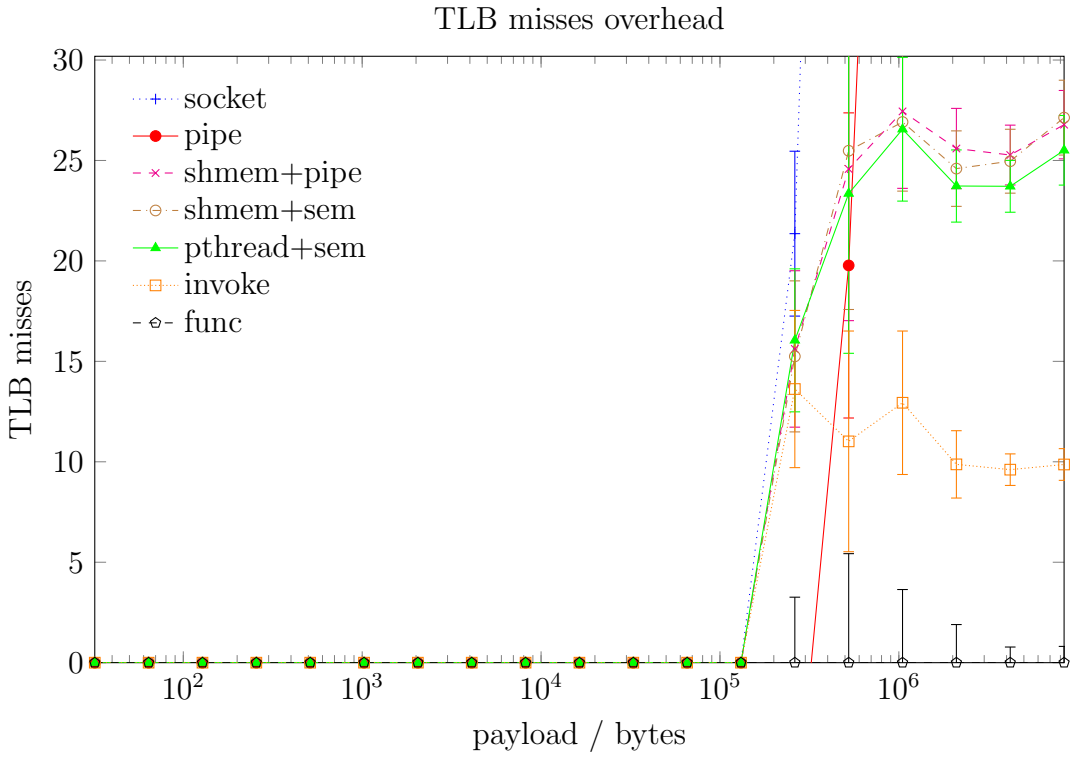
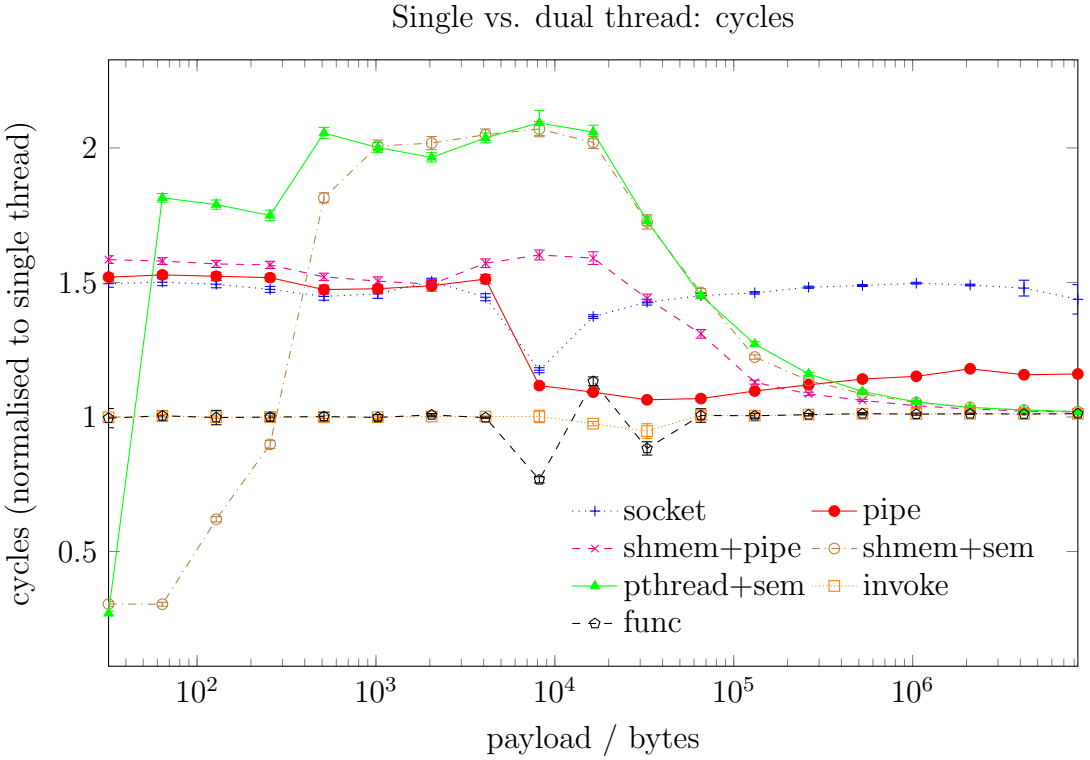
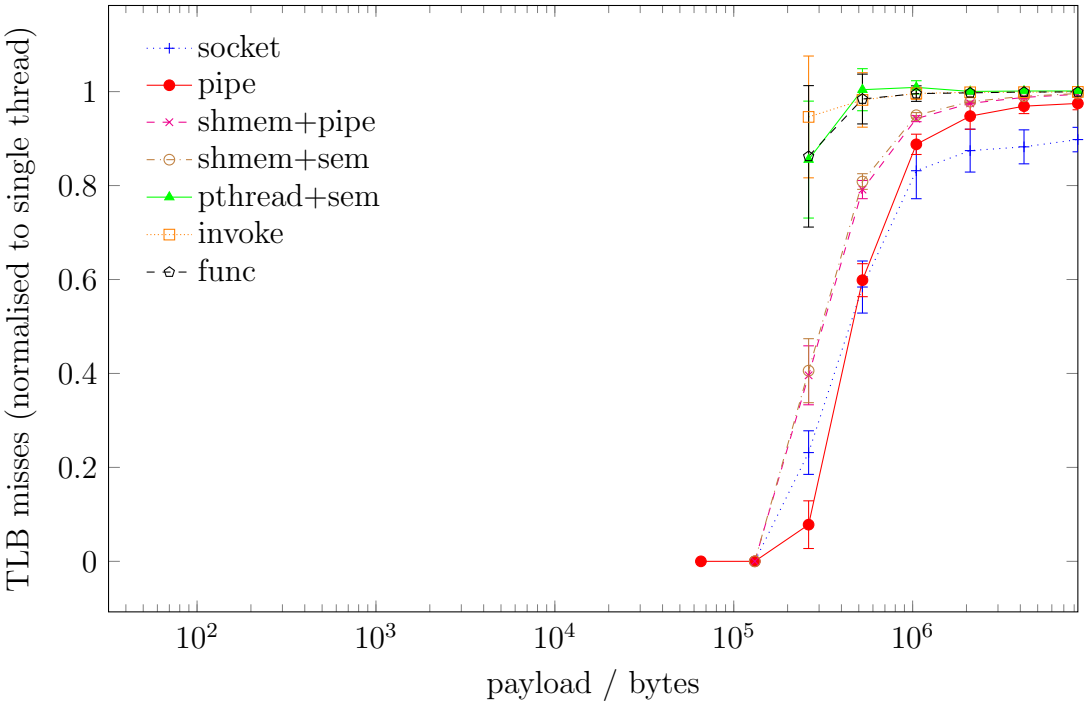


Figure 4.7: Comparison of domain crossing methods – single vs. dual thread

(a) Relative cycle costs



(b) Relative TLB misses



single-threaded operation but with extra fixed overhead which is amortised for the larger payloads .

Note that the number of TLB misses is actually smaller for all multithreaded cases because each hardware thread has a separate 64-entry TLB, meaning that the TLB is effectively doubled in size when using two threads. However performance still decreases overall because the threads share a fixed size L1 TLB-cache as described in Section 3.3 . The trade-off (especially for polling) is different for multicore hardware where threads share fewer hardware resources, but power is still a major consideration . Hardware support for polling, such as `MONITOR`/`MWAIT` or polling detection, may also help but do not address the problem that hardware parallelism and synchronous communication are inherently a bad fit.

I conclude that CHERI provides a much more efficient domain crossing mechanism than process based sandboxing, especially for fine-grained sandboxing where the fixed overheads of IPC are prohibitive. Hardware multithreading can help with shared memory communication for small payloads but the cost and unpredictability in general outweigh the benefits for synchronous workloads, and the performance still does not match a `CCall`. For workloads of over about 2000 cycles CHERI method invocation provides compartmentalisation with efficiency and ease-of-use on a par with a function call.

4.6 Experiment 3: Invoke cost analysis and optimisation

To better understand the costs of a method invocation with `libcheri`, I ran the benchmark with instruction-level tracing on the CHERI prototype, and analysed the traces with respect to a variety of metrics, including cycle and instruction counts . I took multiple samples, discarding runs in which timer interrupts fired, to obtain values for a *best case* domain transition. Table 4.1 shows the instruction count and average cycle costs for different phases of the `CCall` and `CReturn`.

I observed a large cost in clearing registers (including capabilities) that are not used as arguments or return values. This is necessary to prevent leaked data or capabilities that might allow attacks across the interface. I hypothesised that hardware support for clearing registers could reduce this cost with little hardware complexity so I designed a `ClearRegs` instruction for efficiently clearing a given subset of registers. The `ClearRegs` instruction contains a field which specifies a set of 16 target registers (`$r0-$r15`, `$r16-$r31`, `$c0-$c15` or `$c16-$c31`) and a 16-bit immediate mask indicating which of the 16 target registers should be cleared. This is implemented in hardware using a register to record a 'valid' bit for each architectural register. During register read this bit is consulted and, if clear, then zero is returned instead of the value from the register file; when a value is written to the register the valid bit is set. The valid bits can be written efficiently in parallel on `ClearRegs`, making it a single-cycle instruction despite having multiple destinations. CHERI2 does not attempt to forward the value of registers which are cleared, but does detect reads from a register which might be cleared by preceding `ClearRegs` so that the pipeline can be stalled until the `ClearRegs` has been written back. Real code is very unlikely to read from a register which has just been cleared so stalling the pipeline does not cause a performance problem.On a processor with register renaming, such as the original CHERI, `ClearRegs` can be implemented at the renaming stage

Table 4.1: CHERI object-capability invocation costs

Invocation Phase	Software		HW support	
	Inst.	Cyc.	Inst.	Cyc.
caller: setup call, clear unused arg. regs.	29	37	18[b]	28[b]
libcheri: save callee-save regs, push call frame	31	78	31	81
kernel: receive trap, setup kernel	13	26[a]	13	26[a]
kernel: validate ccall args	94	96	79[c]	83[c]
kernel: push trusted stack, unseal `CCall` args.	31	59	31	59
kernel: clear non-arg. regs.	38	38	6[b]	6[b]
kernel: exit handler	7	11	7	11
sandbox: setup sandbox, locate method	31	49[a]	31	51[a]
sandbox: memcpy (payload)	27	78	27	76
sandbox: exit sandbox	3	12	3	12
kernel: receive trap, setup kernel	13	31[a]	13	25[a]
kernel: validate return capability	7	7	7	8
kernel: pop trusted stack	26	60	26	60
kernel: clear non-return registers	54	54	6[b]	6[b]
kernel: exit kernel	7	11	7	11
libcheri: pop call frame, restore registers	29	83[a]	29	81[a]
caller: function return	7	13	7	13
Total	447	743	341	637

[a] Includes overhead of hardware exception (10-15 cycles)
[b] Savings due to hardware `ClearRegs` instruction
[c] Savings due to hardware `CCall` validation

Once implemented in hardware and I modified the kernel to use the instruction and David Chisnall also modified the compiler to use it when preparing for a `CCall`. I found that this saved about 90 instructions and 90 cycles, over 10% of the unoptimised method invoke cost.

Another large cost revealed by tracing is validating that the capability arguments and return values conform with the `CCall` semantics and information-flow policy. Specifically, the policy requires:

1. that the code and data capability arguments to the `CCall` are valid, have matching type fields, appropriate permissions (executable and non-executable respectively) and that the offset field of the code capability is within bounds.

2. that the capability argument registers (`$c3-$c10`) are either untagged (i.e. invalid) or have the global bit set. This is necessary to ensure that 'local' capabilities do not cross sandbox boundaries as described in Section 3.1.7.

The first set of checks can be easily incorporated into the `CCall` instruction in hardware, so I did this and modified the kernel according. This achieved a modest saving of about 15 cycles. The saving would have been larger but the kernel must manually decode the `CCall` instruction to check that the ABI specified argument registers (`$c1` and `$c2`) were used. To simplify the check CHERI has a special register that contains the encoding of the last instruction that caused an exception, avoiding the necessity of loading it from the user space PC with the associated risk of a TLB miss, but the requirement still eliminates some of the saving. Embedding the check into the `CCall` instruction would solve this but would tie the ISA to the ABI, reducing its flexibility.

The check for local capability arguments is costly to perform because it takes a sequence of seven instructions for each of the ten argument registers . Again, an ISA extension could reduce this cost but would be of limited general utility. Other optimisation opportunities exist, but as the validation is specific to CheriBSD's compartment memory model there is a tradeoff between generality and performance.

Other significant costs include saving and restoring callee-save registers (12 general purpose, 11 capability), manipulating the trusted stack, trap overhead (four times for the call and return sequence), and cache and TLB usage.

4.7 Conclusions

These experiments demonstrate that:

1. kernel support for the CHERI extensions adds minimal syscall overhead, and that this overhead is largely dependent on the size of the capability context.

2. the cost of a security domain transition using `CCall` is much less than using process+IPC based sandboxing, being more on a par with a function call in terms of instruction, memory and TLB overhead.

3. simple and cheap hardware support for register clearing can reduce `CCall` overhead, and that further opportunities exist for optimisation should this be necessary given future uses of the CHERI ISA.

CHAPTER 5

Macrobenchmarks

To show that the microbenchmark results in Chapter 4 are reflected in real applications I present the results of running macrobenchmarks: `tcpdump`, and `gzip` and `gif2png` with a compartmentalised `zlib`.

Brooks Davis created a compartmentalised version of the network analysis tool, `tcpdump`, and I present an analysis of the performance of this version using the performance monitoring features of CHERI2. Munraj Vadera's compartmentalised `zlib` is an example of the use of library compartmentalisation.

Similar results were published in our paper at IEEE S&P 2015 [23], but in this chapter I give more extensive results for `cheri_tcpdump`, including analysis of TLB and memory usage which were not available at the time of publication of that paper.

5.1 `tcpdump` compartmentalisation

`tcpdump` contains complex packet dissection code written in C, processes untrusted, potentially malicious network traffic and, due to its need for privileged access to network devices, is frequently run with high privileges such as the `root` user on UNIX systems. It has historically been found to contain many security critical bugs so best-practice dictates that packet capture should be run separately from packet analysis, with analysis being run later as a less privileged user. However, this is unsatisfactory as it not only inconveniences the user, but does not protect the user's account from compromise in the event of an exploitable arbitrary code execution vulnerability.

FreeBSD ships with a version a version of `tcpdump` compartmentalised using Capsicum [33]. This mitigates many problems by dropping the rights to access all resources except the input and output streams during packet processing, but leaves the user vulnerable to attacks which might seek to disable `tcpdump`, for example by crashing or entering an infinite loop (i.e. denial of service), or cause misleading output in order to confuse the user. Such attacks have been used in capture-the-flag competitions to prevent detection or diagnosis of more serious attacks deployed at the same time. In the current implementation Capsicum sandboxing of `tcpdump` is limited to a single sandbox.

The CHERI version of `tcpdump`, `cheri_tcpdump` uses fine-grained compartmentalisation to provide much better protection for the user. It supports several different modes of sandboxing to demonstrate the flexibility and scalability of the sandboxing mechanism. In single sandbox mode `cheri_tcpdump` separates packet processing into a sandbox which

is invoked using the `CCall` mechanism, and resets the sandbox if it causes a protection violation or packet processing takes an excessively long time. This contains any attack to the sandbox, which has no access to outside resources, and prevents denial of service attacks from completely disabling `cheri_tcpdump`. It also supports *temporal* sandboxing by resetting the sandbox after a configurable number of packets have been processed, ensuring that compromised sandboxes are limited in the amount of data they process before being reset. Finally, `cheri_tcpdump` supports separating packets from different flows by taking a hash of the source and destination IP addresses and using this to select from a configurable number of sandboxes. Non-IP traffic is directed to a default 'catchall' sandbox. This means that a compromise of one sandbox is less likely to have access to data from other flows, with this likelihood decreasing as the number of sandboxes increases.

Note that the CHERI sandboxing of `cheri_tcpdump` is in addition to the memory safety benefits afforded by capabilities and also to Capsicum sandboxing, making for an extremely robust application (a security analysis using past vulnerabilities is contained in the IEEE S&P paper).

5.2 `cheri_tcpdump` results

I ran `cheri_tcpdump` on the same quiesced FreeBSD/CHERI2 system as used for the `cheri_bench` benchmarks and measured the initialisation and packet processing costs using CHERI2's user-space performance counters.

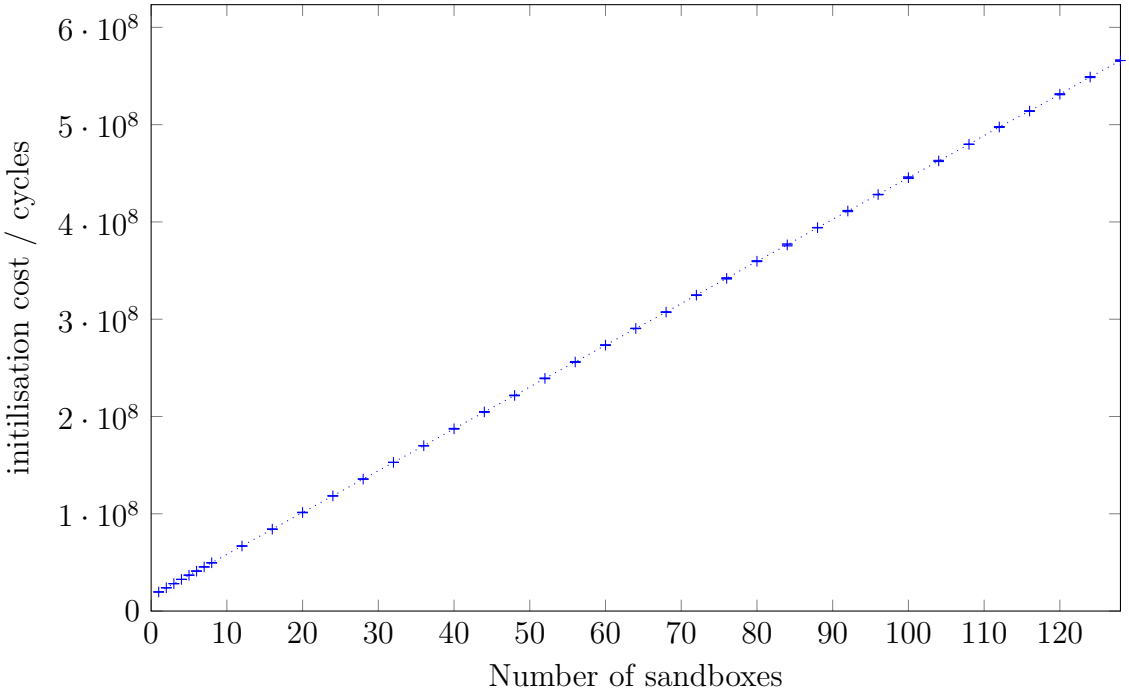
The initialisation phase consists of loading the sandbox *class* from an ELF file using `libcheri` and creating a number of *instances*. Each instance requires allocation of local data and stack memory, and initialisation of local variables using *constructor* functions, therefore there is an expected cycle and memory cost proportional to the number of instances. After initialising the sandboxes `cheri_tcpdump` processes the packets by sending each packet to a sandbox selected based on a hash of the source IP address. For a given number of packets, the time take for the packet processing phase will depend on the overhead associated with sandbox switching, which is determined by the scalability of sandboxing as the number of sandboxes increases.

To measure the scaling properties of CHERI sandboxing I ran `cheri_tcpdump` in its IP sandboxing mode on a trace of 2500 packets and varied the number of sandboxes from one to 128 IP sandboxes in steps of four. There is also an additional non-IP sandbox in each case, although since there was no non-IP traffic in the trace this was not used after initialisation. The trace consisted of a series of short HTTP conversations between varied hosts such that packets were evenly distributed across the IP sandboxes. Packets from different flows were intermixed causing regular switching between sandboxes. I ran the benchmark five times at each point and found the variation between samples to be very low (error bars in the graphs show the standard error in the mean but are too small to be visible in most cases). Textual output was suppressed, leaving just the dissection code running in the sandbox, as otherwise the cost of printing the formatted packet output dwarfed the cost of domain crossing. This allows the scaling behaviour to be observed, and means that in reality sandboxing overhead would be a smaller proportion of the total execution time.

Figure 5.1a shows that the initialisation costs increase as the number of sandboxes increases in a strongly linear fashion. An ordinary least squares (OLS) regression gives an initialisation cost of 4.30 million cycles per sandbox with a y-intercept of 15.2 million

Figure 5.1: `cheri_tcpdump` initialisation costs

(a) `cheri_tcpdump` initialisation costs vs. number of sandboxes



(b) `cheri_tcpdump` maximum resident memory usage vs. number of sandboxes

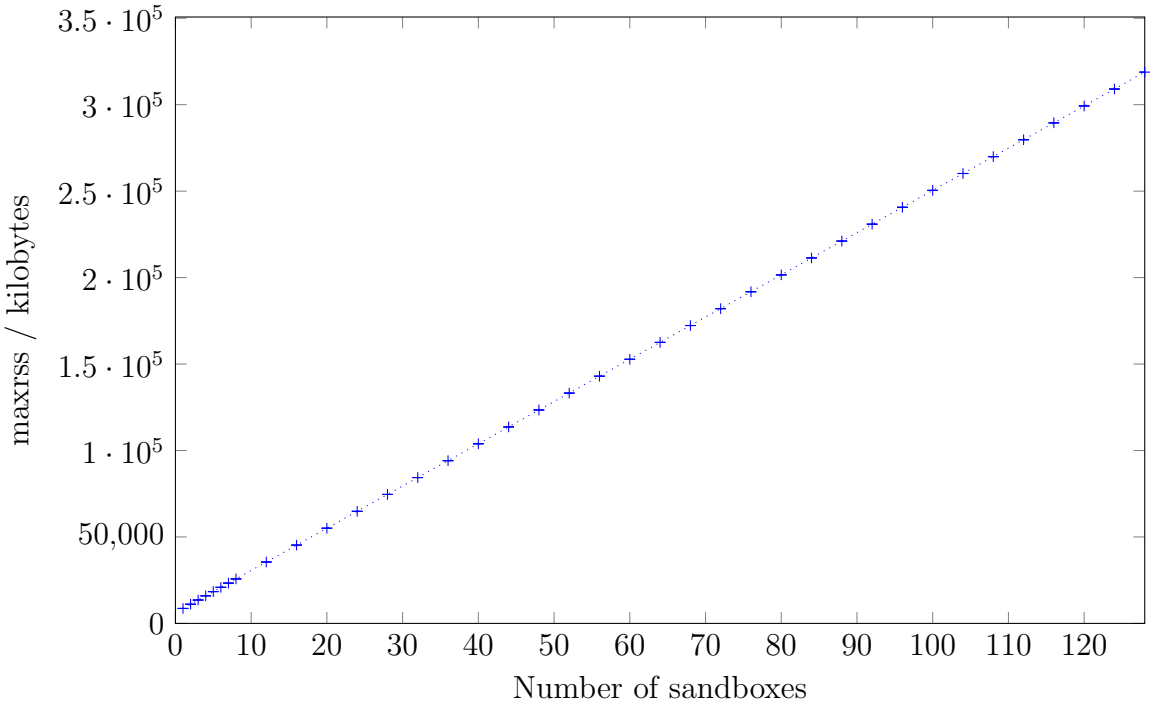
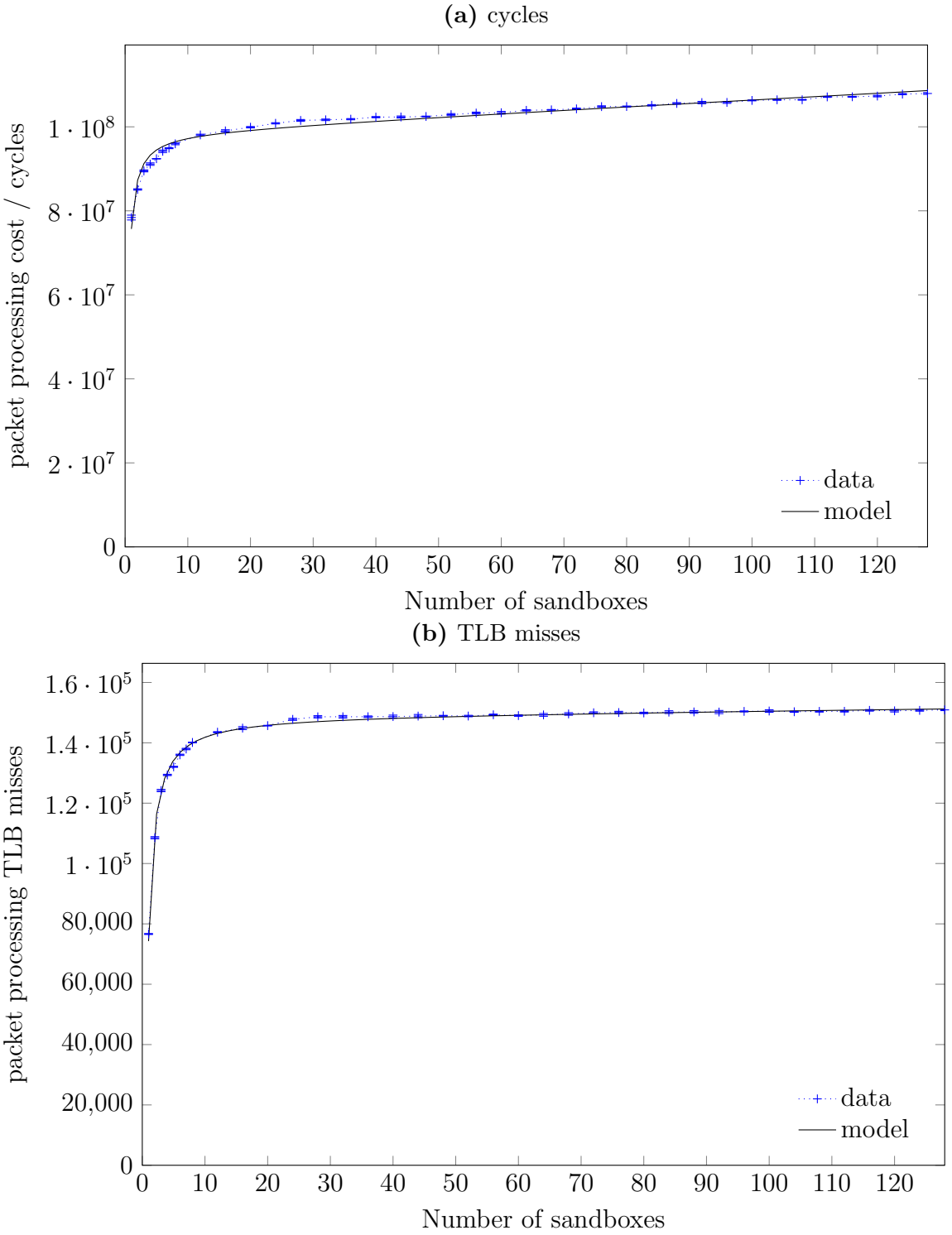


Figure 5.2: `cheri_tcpdump` packet processing costs vs. number of sandboxes with fitted model



cycles corresponding to the cost of initialising `cheri_tcpdump` with just a single sandbox – the 'catchall' for non-IP traffic. The statistical significance of the fit is extremely high ($R^2 > 0.9999$). The other metrics I measured, such as TLB and cache misses, show identical linear initialisation costs. Each sandbox instance requires memory to be allocated for sandbox local data and initialised so this linear dependence is not surprising.

I also measured the maximum resident memory as reported by `getrusage` at the end of each `cheri_tcpdump` execution. Figure 5.1b shows that each sandbox consumes a fixed amount of additional memory, with an OLS regression revealing this to be about 2.4 MiB per sandbox on top of the initial 6.2 MiB used by the single sandbox version.

Figures 5.2a and 5.2b show the number of cycles and TLB misses taken to process all 2500 packets after initialisation. There is a sharp initial rise in both metrics followed by a gradual linear increase (I took extra samples in the initial region to show the curve). This can be explained by the fact that the probability that a packet will use the same sandbox as the previous packet is roughly $1/N$ where N is the number of sandboxes (assuming uniformly randomly distributed packets). Each sandbox has local data which must be faulted in when the sandbox is called if it is not already present in the caches and TLB, leading to greater costs for high numbers of sandboxes due to reduced temporal locality. Therefore I attempted to fit a model $y = \beta_0 + \beta_1 x + \beta_2/x$ where the linear $\beta_1 x$ term accounts for the first invocation of a sandbox which will lead to compulsory misses and the β_2/x term is expected to be negative to account for cost savings when there is some sandbox locality. Fitting this model using OLS regression explains much of the variation with adjusted R^2 of 0.976 (cycles) and 0.995 (TLB misses), but analysis of the residuals using the Jarque-Bera statistic shows that they are not normally distributed, indicating that the model is not perfect. This is probably because the locality is not modelled precisely by the β_2/x term – the real curve will depend on the amount of locality in the trace and the cache and TLB footprint of the sandbox as there is probably data from more than just the most recently used sandbox in the caches. Nevertheless this simple model describes the data to some extent and tells us one important thing: for large numbers of sandboxes the cost is dominated by the linear term which has a very shallow gradient, meaning CHERI can scale to large numbers of sandboxes efficiently.

5.3 `gzip`/`zlib` compartmentalisation

The widely used `zlib` compression library has a history of security vulnerabilities and, since it is common to decompress untrusted data downloaded over the internet, this makes it a good candidate for compartmentalisation. By taking the library compartmentalisation approach described in Section 3.8.2 all applications that use `zlib` can benefit from compartmentalisation whilst only making the required code changes once.

To test this approach Munraj Vadera compartmentalised the `gzip` compression tool, which uses `zlib`, using four different approaches: the product of two compartmentalisation technologies (Capsicum and CHERI) and two strategies which place the security boundary in different locations. In the first strategy the application is divided into two parts where the privileged part is responsible for parsing arguments and opening input and output files and the second part performs compression using delegated file descriptors. The second strategy is library compartmentalisation where a `zlib` compatible wrapper forwards library calls to a sandbox implemented either using Capsicum or CHERI.

Capsicum provides an example of using traditional process based isolation techniques.

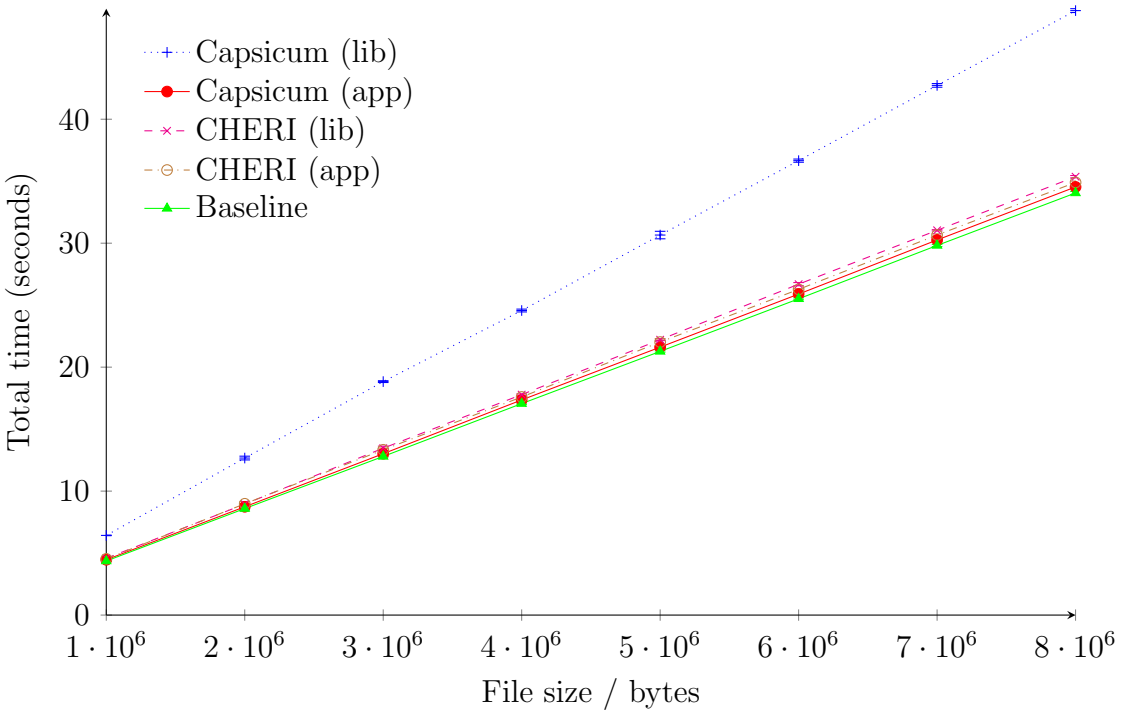


Figure 5.3: Compression time for `gzip` with various sandboxing mechanisms

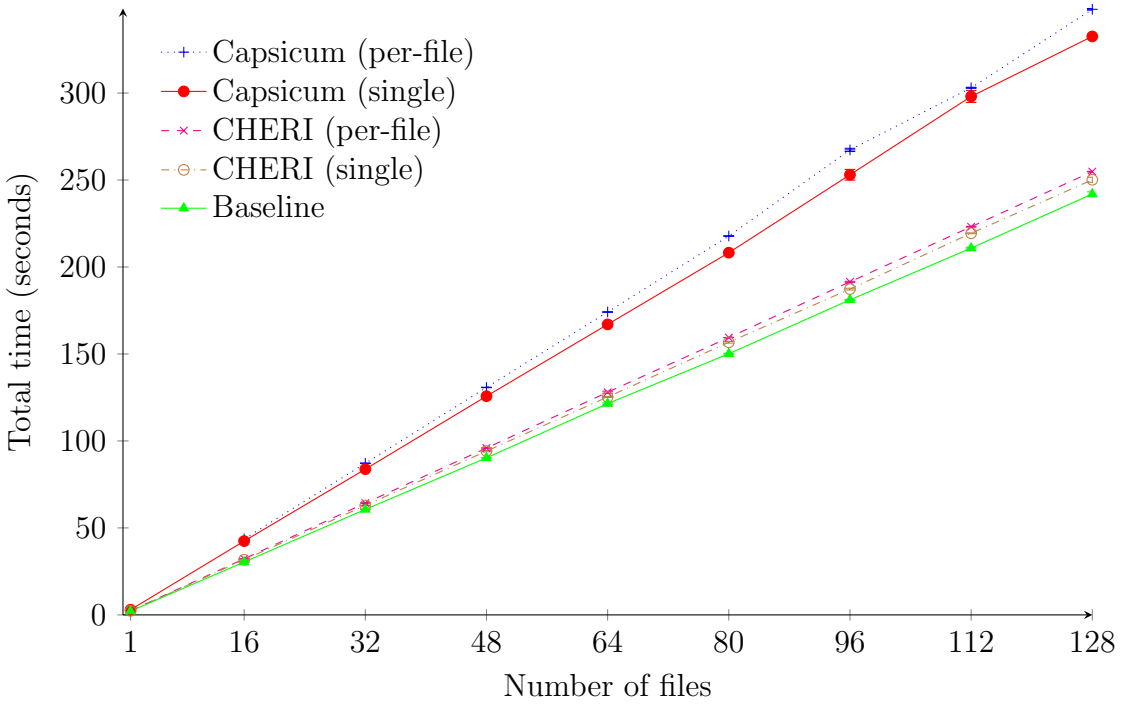


Figure 5.4: Sandbox creation overhead: time taken to compress varying numbers of files of size 500,000 bytes with different `zlib` implementations

It is well suited to the application compartmentalisation strategy because `gzip` has a simple program structure which can be divided into two phases: after opening input and output file descriptors `gzip` calls the `cap_enter` system call to drop ambient privileges, retaining access to only the open input and output files for the risky compression phase. This incurs a small fixed overhead vs. uncompartmentalised `gzip` as shown in Figure 5.3, which shows results from running on the original CHERI as used in the IEEE S&P paper.

However, the `zlib` API is difficult to compartmentalise at the library level because it passes buffers via pointers (a common approach in performance sensitive data processing libraries). With Capsicum this requires data to be copied across the compartment boundary using IPC, incurring linear overhead in the size of the data. This leaves developers using Capsicum with a choice between modifying each application on an individual basis, with varying degrees of difficulty, and the powerful library compartmentalisation approach which suffers from significant performance overhead.

CHERI, on the other hand, is well-suited to both compartmentalisation strategies owing to its ability to delegate memory regions via capabilities. Thus we see in Figure 5.3 that both CHERI strategies achieve performance close to the baseline, albeit with some overhead due to the `CCall` on each call to `deflate` and due to using capabilities for all pointers within the sandbox. Nevertheless the CHERI overhead is low in both cases considering the security benefits afforded by both compartmentalisation and pointer safety.

5.4 `gzip` per sandbox costs

The cost of sandboxing is not limited to communication overheads, but must also take into account the cost of sandbox creation and initialisation. In the case of process based compartmentalisation this can be high as each sandbox requires and expensive UNIX `fork`. To demonstrate this the compartmentalised `zlib` versions were modified to use a new sandbox for each compression context (i.e. file in the case of `gzip`). This is desirable to prevent the possibility of information leakage caused by reusing the same sandbox to compress multiple files from different sources.

Figure 5.4 shows the results for compressing varying numbers of files using either single-sandbox or per-file library compartmentalisation with Capsicum and CHERI. Both Capsicum and CHERI incur some per sandbox overhead, but the Capsicum overhead is much more significant, with CHERI remaining within 5% of the baseline . This confirms the scalability of CHERI sandboxing in contrast to process-based methods.

5.5 `zlib`/`gif2png` library compartmentalisation example

The major advantage of library compartmentalisation is that all users of a library can benefit without even needing to be re-compiled. The `gif2png` image conversion utility uses `zlib` via the `libpng` image compression library. Figure 5.5 shows that the performance of `gif2png` when linked to the CHERI compartmentalised `zlib` is indistinguishable from the uncompartmentalised version for three different sizes of input, being within the experimental margin of error . This is because `gif2png` is a heavily compute bound application and therefore the overhead of CHERI compartmentalisation is very small relative to the overall execution time.

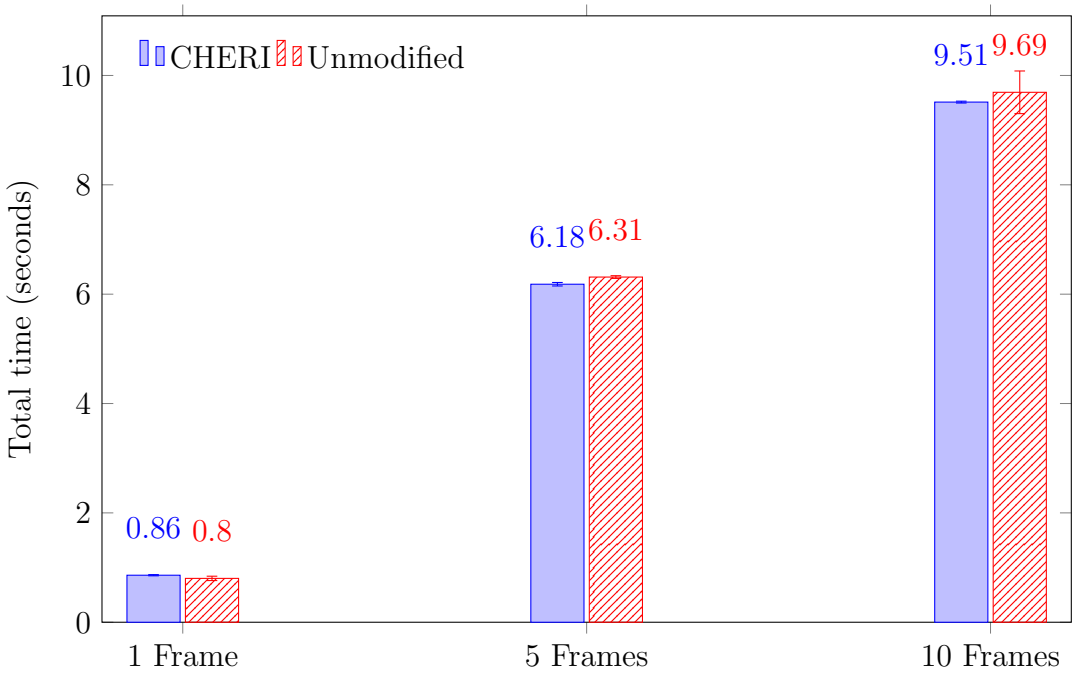


Figure 5.5: `gif2png` with unmodified and CHERI `zlib` implementations

5.6 Conclusions

The benchmarks in this chapter show that CHERI compartmentalisation can be practically applied to real applications and that it is possible to scale to large numbers of sandboxes efficiently. The main obstacle to scalability in `cheri_tcpdump` is the memory footprint of the sandboxes, which is inflated by the large number of global variables which need to be copied for each new sandbox instance, increasing the initialisation and memory usage costs. This could be optimised as many of the global variables are not needed, but the important thing to note is that even with large numbers of sandboxes the performance is acceptable and appears to scale well.

The `zlib` example demonstrates the ability of CHERI to efficiently compartmentalise library code even in the presence of a buffer-oriented API which is a significant obstacle to using state-of-the-art virtual memory based Capsicum compartmentalisation.

chapter 6

Conclusions and Future Work

This dissertation has provided a thorough performance evaluation of the compartmentalisation features of the CHERI ISA. I have demonstrated that these features are efficient and can be used to compartmentalise existing applications with reasonable programming effort and acceptable performance costs. Furthermore, I have shown that domain crossing can be optimised with hardware support for clearing capability and integer registers.

6.1 Conclusions

I have demonstrated that hardware support for compartmentalisation can enable fine-grained and efficient application compartmentalisation, an extremely desirable feature in the modern, security conscious computing environment. I have described existing approaches to this problem and why they fail to meet requirements in terms of performance, security and ease of use, either because of the requirement to use new, incompatible languages and programming environments, or due to reliance on coarse-grained and slow memory protection hardware available in current commodity hardware.

The `cheri_bench` microbenchmark I created simultaneously illustrates the deficiency of virtual memory based domain crossing techniques and highlights the efficiency of the CHERI `CCall` mechanism, both in terms of domain crossing latency and data transfer costs. By obviating the need to use virtual memory for isolation CHERI creates the possibility for fine-grained compartmentalisation with efficient and flexible data sharing. It reduces the pressure on the Translation Lookaside Buffer by ensuring a one-to-one relation between data pages and TLB entries, rather than the one-to-many relation required to implement shared memory.

By analysing the performance of `cheri_tcpdump`, a compartmentalised version of the popular `tcpdump` network packet analyser, I have shown that CHERI compartmentalisation has acceptable overhead when applied to real applications, and that it can scale to a large number of sandboxes for increased security. This also demonstrates the feasibility of porting existing C-language applications to CHERI using the advanced compiler support for capabilities and security domain crossing calls.

I conclude that hardware support for compartmentalisation is a compelling feature of the CHERI architecture, perfectly complementing its already excellent memory safety benefits. I therefore encourage future computer architects to consider fine-grained memory protection and compartmentalisation as essential to any modern CPU design.

6.2 Engineering Contributions

The CHERI project is a complete research platform from top to bottom, comprising the hardware (FPGA prototype), compiler (clang/llvm + CHERI extensions), operating system (CheriBSD), libraries and applications as well as a range of supporting infrastructure such as tests, debugging, tracing and binary tools. As such it is necessarily the collaborative work of many people, but I personally have made some important contributions:

- I implemented most of the CHERI2 multithreaded CPU prototype, an alternative to the original CHERI with a number of advantages in terms of coding style and features for benchmarking. In particular it is coded in a way to make it amenable to formal analysis, avoiding Bluespec features such as `RWires` (raw wires) which would hamper such research. Nirav Dave is undertaking this analysis but it is not the focus of my efforts or this dissertation.

- In common with all large projects CHERI requires a lot of testing infrastructure to ensure everything works together. During development I added assembly level tests to verify various features of the ISA implementation, including a simple test generator to produce large numbers of tests comparing CHERI and CHERI2 to an existing MIPS simulator, `gxemul`.

- I added features to CHERI2 to aid with performance analysis, such as performance counters and cycle-accurate instruction tracing. I also wrote a program to print and analyse traces, which I used to produce some of the graphs in this dissertation. (Section 3.4)

- In the process of running and analysing benchmarks I added many features to CHERI2 to make it a more realistic and powerful platform. I discovered that a disadvantage of the original CHERI for benchmarking is its use of direct-mapped caches for the instruction, data and TLB caches, which introduced undesirable performance artefacts due to conflicts. Therefore I implemented set-associative level one caches and a fully-associative TLB in CHERI2 making it more realistic and repeatable for benchmarking. (Section 3.3)

- I implemented `cheri_bench`, a microbenchmark to study different ways of creating and communicating between sandboxes, including commonly used process based isolation and CHERI's innovative `CCall` mechanism. (Section 4.1)

- After analysing the complete `CCall` sequence using tracing I observed that register clearing represented a significant part of the total cost. Therefore I designed and implemented the `ClearRegs` instruction and modified CheriBSD to use it, demonstrating that CHERI domain crossing is amenable to further optimisation using simple hardware support. (Section 4.6)

- I also used CHERI2 to analyse the performance of a compartmentalised version of `tcpdump`. I found that there is a significant initial performance overhead explained by reduced locality as a result of rapid sandbox switching, but that for large numbers of sandboxes the cost scales linearly with a low incremental cost. (Chapter 5)

- I was concerned that the lack of thread-level parallelism in CHERI2 would unfairly disadvantage the process-based compartmentalisation benchmarks, so I implemented a simple form of hardware multithreading to emulate multicore behaviour. In the end I found that multithreading did not help most benchmarks due to their synchronous nature and the software overhead (in the form of inter-thread interrupts) required for TLB synchronisation between the threads. I did discover that multithreading helped when communicating via shared memory using polling, but only under certain circumstances (very small payloads), leading me to conclude that multithreading is not a reliable way to accelerate sandboxing using traditional methods. (Section 4.5)

- To support the multithreading work I designed and implemented a programmable interrupt control (PIC), which was necessary to perform inter-thread interrupts for use by CheriBSD. The same PIC is currently being used in efforts to develop a multicore CHERI processor and the existence of the multithreaded version was helpful when developing FreeBSD support for symmetric multiprocessing on CHERI.

6.3 Future Work

The wide-reaching implications of fine-grained memory protection and compartmentalisation are only just beginning to be explored, and there is a great deal of work which can still be done, particularly if these features are to be adopted by existing commercial processors.

6.3.1 Domain crossing optimisations

CHERI's `CCall` instruction provides an efficient means of crossing security domain boundaries, but the cost is still non-trivial when compared to a standard function call, slightly restricting the uses to which it can be put. For example replacing every method invocation in a object-oriented language implementation such as C++ or Objective-C is probably not feasible with the current costs.

My evaluation revealed some opportunities to optimise the existing domain crossing code. In particular, I demonstrated savings through the use of simple hardware checks in the `CCall` instruction and added a `ClearRegs` instruction for efficient register invalidation, but there are other opportunities which have yet to be evaluated.

For example, validating that each of the capability arguments is either global or tagged invalid requires a sequence of seven instructions using the current ISA and this could easily be simplified through the addition of a new instruction. I did not implement this, however, due to its limited generality and my concern that the use of local capabilities might change under different sandboxing schemes (as noted this in Section 3.1.7 this has already happened once during development).

Another potential optimisation would be to eliminate the trap handler altogether by replacing the `CCall` trap with a microcoded instruction or even a form of PALCode as used on the DEC Alpha [91]. This would perform the necessary trusted operations, such as argument validation and trusted stack manipulation, without entering the kernel, avoiding the overhead of taking a trap and the instruction cache footprint of the trap handler. However, this would risk premature ossification of the `CCall` semantics and would be of

uncertain benefit so it makes more sense to retain the flexibility of a software trap for now.

More generally, it should be observed that CHERI is an experimental platform and is thus biased more towards flexibility and generality than it is optimised for a specific usage scenario. This is particularly reflected in the large number of capability registers and their large size. Ongoing work is investigating the possibility of *compressed* 128-bit capabilities, with the potential to reduce both hardware and runtime costs. Further to this, any commercial adoption of CHERI-like features would have to consider the optimal number of capability registers to balance the hardware costs and context switching overhead against the need to prevent excessive spilling of capability registers to the stack and for passing arguments. Both of these optimisations would shrink the size of the capability context, reducing the overhead of domain crossing further.

Given the synergy between threads, capabilities and security domains, another possible avenue for exploration would be to add support for efficient, synchronous communication between hardware threads. For example, a synchronous call instruction could suspend the current hardware thread and resume a waiting one, effecting a domain transition without the need to load or store any capability state. Such mechanisms have been proposed before to enable efficient parallelism, such as in the XMOS XS1 architecture [92] and the Mamba processor [93], but could be particularly attractive in the context of security domain crossing.

However, until the consequences of fine-grained compartmentalisation are further studied, it is not clear how much optimisation of domain crossing is necessary. There are many potential uses for the CHERI compartmentalisation features and these will have different requirements depending on the nature of the sandboxes, their size and the relationships between them.

6.3.2 Alternative uses for compartmentalisation features

More work is required to investigate other potential applications of the CHERI compartmentalisation features. For example, the current sandboxing model used in FreeBSD is limited to user-space applications, but capabilities could also be used in the kernel to separate different modules from each other, perhaps returning to the microkernel ideas that were largely abandoned due to the performance issues which CHERI addresses.

Research is already ongoing on the possibility of using capabilities for the system call interface between user-space and the kernel. This would remove the need to proxy system calls from sandboxes via a more privileged 'system' sandbox, allowing sandboxes to directly invoke system calls, passing buffers via capabilities as arguments and potentially using capabilities as tokens to authorise use of a particular system calls.

Another use would be in adapting a high-level language, such as C++ or Objective-C to enforce language-level compartments, i.e. objects, using the CHERI hardware primitives, potentially simplifying and strengthening the implementation.

Bibliography

[1] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security Privacy, IEEE*, 9(3):49–51, May 2011.

[2] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[3] Dominic Rushe. Hackers who targeted Sony invoke 9/11 attacks in warning to moviegoers. *Guardian Newspaper*, 16 December 2014.

[4] Oxford Economics. Cyber-attacks: Effects on UK Companies, July 2014. Report commissioned by UK Centre for the Protection of National Infrastructure.

[5] Out of copyright image from article The White Star liner Titanic. *Engineering Journal*, 91, May 1911.

[6] Stefan Frei. Vulnerability threat trends. Technical report, NSS Labs, 2013.

[7] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[8] Paul A. Karger. Limiting the damage potential of discretionary trojan horses. In *IEEE Symposium on Security and Privacy*, pages 32–37, 1987.

[9] Herman Miller, John McIntire, and James Southerland. Damage controlman nonresident training course, chapter 3: Ship compartmentation and watertight integrity. Technical Report NAVEDTRA 14057, United States Navy, 2001.

[10] Paul A. Karger, Usaf Roger, and R. Schell. Multics security evaluation: Vulnerability analysis. Technical Report ESD-TR-74-193, HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/karg74.pdf, 1974.

[11] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972.

[12] Maurice Wilkes. Operating systems in a changing world. *SIGOPS Oper. Syst. Rev.*, 28(2):9–21, April 1994.

[13] Neils Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003.

[14] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232, New York, NY, USA, 2009. ACM.

[15] 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, pages 203–216, New York, NY, USA, 1993. ACM.

[16] B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93, May 2009.

[17] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[18] Emmett Witchel and Krste Asanovic. Hardware Works, Software Doesn't: Enforcing Modularity with Mondriaan Memory Protection. In *HotOS*, pages 139–144, 2003.

[19] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. ACM.

[20] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 190–202, Washington, DC, USA, 2014. IEEE Computer Society.

[21] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. Codoms: Protecting software with code-centric memory domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 469–480, Piscataway, NJ, USA, 2014. IEEE Press.

[22] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, **Robert Norton**, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st annual international symposium on computer architecuture*, pages 457–468. IEEE Press, 2014.

[23] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, J. Anderson, D. Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Stephen J. Murdoch, **Robert Norton**, Mike Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37, May 2015.

[24] Jonathan D. Woodruff. CHERI: A RISC capability machine for practical memory safety. Technical Report UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014.

[25] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, **Robert Norton**, and Stacey Son. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory, September 2015.

[26] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, **Robert Norton**, and Michael Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, April 2015.

[27] Brooks Davis, **Robert Norton**, Jonathan Woodruff, and Robert NM Watson. How FreeBSD Boots: a soft-core MIPS perspective. In *Proceedings of AsiaBSDCon 2014*, March 2014.

[28] Robert J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[29] Bill Cheswick. An Evening with Berferd in which a cracker is Lured, Endured, and Studied. In *Proceedings of the Winter USENIX Conference*, pages 163–174, 1992.

[30] A.G. Tucker, J.T. Beck, D.S. Comay, A.D. Gabriel, O.C. Leonard, and D.B. Price. Global visibility controls for operating system partitions, October 14 2008. US Patent 7,437,556.

[31] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[32] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[33] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[34] David E. Bell and Leonard J. Lapadula. Secure Computer System: Unified Exposition and MULTICS Interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, March 1976.

[35] Mick Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.*, 2006(148):13–, August 2006.

[36] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux security module. Technical Report 01-043, NAI Labs, December 2001.

[37] M Accetta, R Baron, W Bolosky, D Golub, R Rashid, A Tevanian, M Young, R Baron, D Black, W Bolosky, et al. Mach: A new kernel foundation for unix development. In *Summer Conference Proceedings 1986*, volume 4, pages 64–75. USENIX Association, 1986.

[38] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. *SIGOPS Oper. Syst. Rev.*, 27(5):120–133, December 1993.

[39] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[40] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, December 1999.

[41] Alan Au and Gernot Heiser. L4 user manual. Technical Report UNSW-CSE-TR-9801, University of New South Wales, School of Computer Science and Engineering, 1998.

[42] Jochen Liedtke, Kevin Elphinstone, Sebastian Schonberg, Hermarill Härtig, Gernot Heiser, Nahina Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Operating Systems, 1997., The Sixth Workshop on Hot Topics*, pages 28–31. IEEE, 1997.

[43] Jonathan S Shapiro, David J Farber, and Jonathan M Smith. The measured performance of a fast local IPC. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems*. IEEE, 1996.

[44] C. McCurdy, A.L. Cox, and J. Vetter. Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 95–104, April 2008.

[45] Broderick Ian Aquilino. Flashback OS X malware. In *Proceedings of Virus Bulletin Conference*, pages 102–114, September 2012.

[46] Peter W. Madany. JavaOS™: A standalone Java™ environment. JavaSoft White Paper, May 1996.

[47] Tom Saulpaugh and Charles A Mirho. *Inside the JavaOS operating system*. Addison-Wesley, 1999.

[48] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[49] Christopher Small and Christopher Small. MiSFIT: A tool for construction safe extensible C++ systems. In *Third Conference on Object-Oriented Technologies and Systems*, 1997.

[50] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[51] Stephen Mccamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *In 15th USENIX Security Symposium (2006*, pages 209–224, 2006.

[52] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[53] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 45–58, New York, NY, USA, 2009. ACM.

[54] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[55] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, February 2005.

[56] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the jvm. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[57] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[58] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture support for single address space operating systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 175–186, New York, NY, USA, 1992. ACM.

[59] John Wilkes and Bart Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. Technical Report HPL-92-55, Hewlett-Packard Laboratories, 1992.

[60] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 12 Crosby Dr., Bedford, MA, USA, 1984.

[61] Alastair J. W. Mayer. The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10, June 1982.

[62] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[63] P.G. Neumann, L. Robinson, K.N. Levitt, R.S. Boyer, and A.R. Saxena. A provably secure operating system. Technical report, Computer Science Laboratory SRI International, Menlo Park, California, 13 June 1975.

[64] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical report, Computer Science Laboratory SRI International, Menlo Park, California, 11 February 1977.

[65] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[66] R.J. Feiertag and P.G. Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979. `http://www.csl.sri.com/neumann/psos.pdf`.

[67] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society.

[68] DM England. Capability concept mechanism and structure in system 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, 1974.

[69] R. M. Needham and R. D.H. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 1–10, New York, NY, USA, 1977. ACM.

[70] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[71] V. Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 245–252. ACM, 1980.

[72] C.B. Hunter, J.F. Ready, and E. Farquhar. *Introduction to the Intel iAPX 432 architecture*. Reston Pub. Co.(Reston, Va.), 1985.

[73] David A Patterson and Carlo H Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 443–457. IEEE Computer Society Press, 1981.

[74] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *SIGARCH Comput. Archit. News*, 43(1):117–130, March 2015.

[75] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *20th ACM Conference on Computer and Communications Security*, November 2013.

[76] U. Dhawan, A. Kwon, E. Kadric, C. Hritcu, B.C. Pierce, J.M. Smith, A. DeHon, G. Malecha, G. Morrisett, T.F. Knight, A. Sutherland, T. Hawkins, A. Zyxnfryx,

D. Wittenberg, P. Trei, S. Ray, and G. Sullivan. Hardware support for safety interlocks and introspection. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*, Sept 2012.

[77] Kanad Ghose and Pavel Vasek. A fast capability extension to a RISC architecture. In *22rd EUROMICRO Conference '96, Beyond 2000: Hardware and Software Design Strategies, September 2-5, 1996, Prague, Czech Republic*, page 606, 1996.

[78] Pavel Vasek and Kanad Ghose. A comparison of two context allocation approaches for fast protected calls. In *Proceedings of the Fourth International on High-Performance Computing, HiPC 1997, Bangalore, India, 18-21 December, 1997*, pages 16–21, 1997.

[79] Building a secure system using TrustZone technology. Technical Report PRD29-GENC-009492CPRD29-GENC-009492C, ARM Limited, April 2009.

[80] Rob Coombs. Securing the future of authentication with ARM TrustZone-based trusted execution environment and Fast Identity Online (FIDO). ARM White Paper, May 2015.

[81] Intel software guard extensions programming reference. Technical Report 329298-002US, Intel Corporation, October 2014.

[82] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. *SIGPLAN Not.*, 43(3):103–114, March 2008.

[83] Intel Plc. Introduction to Intel Memory Protection Extensions. `http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, July 2013.

[84] C.W. Otterstad. A brief evaluation of Intel iMPX. In *Systems Conference (SysCon), 2015 9th Annual IEEE International*, April 2015.

[85] Daniel L. Rosenband and Arvind. Modular scheduling of guarded atomic actions. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 55–60, New York, NY, USA, 2004. ACM.

[86] Daniel L. Rosenband and Arvind. Hardware synthesis from guarded atomic actions with performance specifications. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, pages 784–791, Washington, DC, USA, 2005. IEEE Computer Society.

[87] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '00, pages 511–519, Piscataway, NJ, USA, 2000. IEEE Press.

[88] FPGA architecture. Technical Report WP-01003-1.0, Altera Corporation, July 2006.

[89] R. P. Weicker. Dhrystone benchmark: Rationale for version 2 and measurement rules. *SIGPLAN Not.*, 23(8):49–62, August 1988.

[90] Alan R. Weiss. Dhrystone benchmark: History, analysis, scores and recommendations. White paper, October 2002.

[91] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Newton, MA, USA, 1992.

[92] David May. XMOS XS1 architecture, version 8.7. White Paper, July 2008.

[93] G.A. Chadwick and S.W. Moore. Mamba: A scalable communication centric multi-threaded processor architecture. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 277–283, Sept 2012.