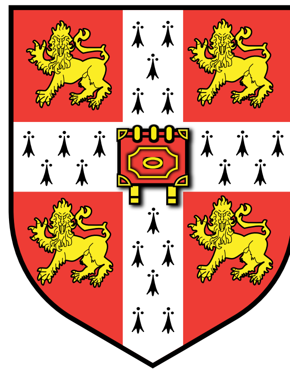
Capability Memory Protection for Embedded Systems

Hongyan Xia

University of Cambridge

Department of Computer Science and Technology



Hughes Hall

May 2019

This dissertation is submitted for the degree of Doctor of Philosophy

View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

Capability Memory Protection for Embedded Systems

Hongyan Xia

Abstract:

This dissertation explores the use of capability security hardware and software in real-time and latency-sensitive embedded systems, to address existing memory safety and task isolation problems as well as providing new means to design a secure and scalable real-time system. In addition, this dissertation looks into how practical and high-performance temporal memory safety can be achieved under a capability architecture.

State-of-the-art memory protection schemes for embedded systems typically present limited and inflexible solutions to memory protection and isolation, and fail to scale as embedded devices become more capable and ubiquitous. I investigate whether a capability architecture is able to provide new angles to address memory safety issues in an embedded scenario. Previous CHERI capability research focuses on 64-bit architectures in UNIX operating systems, which does not translate to typical 32-bit embedded processors with low-latency and real-time requirements. I propose and implement the CHERI CC-64 encoding and the CHERI-64 coprocessor to construct a feasible capability-enabled 32-bit CPU. In addition, I implement a real-time kernel for embedded systems atop CHERI-64. On this hardware and software platform, I focus on exploring scalable task isolation and fine-grained memory protection enabled by capabilities in a single flat physical address space, which are otherwise difficult or impossible to achieve via state-of-the-art approaches. Later, I present the evaluation of the hardware implementation and the software run-time overhead and real-time performance.

Even with capability support, CHERI-64 as well as other CHERI processors still expose major attack surfaces through temporal vulnerabilities like use-after-free. A naïve approach that sweeps memory to invalidate stale capabilities is inefficient and incurs significant cycle overhead and DRAM traffic. To make sweeping revocation feasible, I introduce new architectural mechanisms and micro-architectural optimisations to substantially reduce the cost of memory sweeping and capability revocation. Another factor of the cost is the frequency of memory sweeping. I explore tradeoffs of memory allocator designs that use quarantine buffers and shadow space tags to prevent frequent unnecessary sweeping. The evaluation shows that the optimisations and new allocator designs reduce the cost of capability sweeping revocation by orders of magnitude, making it already practical for most applications to adopt temporal safety under CHERI.

Acknowledgements

First, I would like to thank my supervisor, Professor Simon Moore, for his patience, technical expertise and great guidance throughout my PhD. I never imagined that I could accomplish what I have actually achieved in the past years. Looking back, I am sincerely grateful to Simon whose supervision makes the entire process exciting, rewarding, challenging, and most of all, enjoyable.

I would also like to thank Robert Watson for his insights in software, kernels and operating systems. With his supervision, I have been exposed to the art of software-hardware co-design, which successfully transformed some of my work from what I could only call engineering effort, into comprehensive and more sophisticated research topics.

I am also grateful to Jonathan Woodruff, who shows great wisdom in CPU architecture and design. He also shows unparalleled patience compared with most human beings I have seen. I often feel guilty that a lot of his time was spent on sitting beside me helping me understand CHERI concepts and CPU micro-architecture instead of having biscuits and tea or on other meaningful activities. Hopefully, I am now repaying his kindness with my own contributions to the project.

I also need to thank Alexandre Joannou for his knowledge and help on various fronts and his work style. Particularly, he is an interesting colleague to work with, and his obsession with high-level abstractions often sparks long and meaningful discussions. Sometimes his design philosophy is something I would be thankful for several months later when I see the code evolve into a modular, configurable entity instead of being filled with ugly hacks and hardcoded values. Working with him in the past years has always been a pleasure.

I need to thank other members in the team, including David Chisnall, Robert Norton, Theo Markettos, Robert Kovacsics, Marno van der Maas, Lucian Paul-Trifu, Nathaniel Filardo, Lawrence Esswood, Peter Rugg and so forth, for all the help and collaboration in multiple areas and projects. It always reminds me of how wonderful it is to work within an active, open-minded, innovative and motivated research team.

Outside my research, I have been enjoying my time with Yimai Fang, Fengyuan Shi, Ruoyu Zhou, Menglin Xia, Dongwei Wang, Jiaming Liang, Meng Zhang, Zheng Yuan... in countless punting trips, excursions, Nintendo Switch nights, festival celebrations, formal hall dinners, which gave me some of the best memories I have had.

I thank my parents for their support for my PhD and for everything. Frankly, it is not possible to express my eternal gratitude here with just words. What I do know is that whenever I am lost, you are there, strong and steady. Wish you guys health, sincerely.

Finally, I must thank Xizi Wei for just being in my life. You are a food innovator, a good listener, a healthy lifestyle enforcer, a cat lover, a Machine Learning expert, a perfectionist, a bookworm, and above all, a wonderful human being. I hope I can achieve this many positive labels in your mind, and I will try.

Contents

List of Figures **13**

List of Tables **15**

1 Introduction **17**

1.1 Contributions . . . 18

1.2 Publications . . . 19

1.3 Dissertation overview . . . 20

2 Background **21**

2.1 Introduction to embedded systems . . . 21

2.2 The need for memory safety . . . 23

2.3 State-of-the-art memory protection . . . 24

2.3.1 Desktop systems . . . 24

2.3.2 Searching for novel solutions for embedded systems . . . 24

2.4 Case studies . . . 27

2.4.1 The Broadcom Wi-Fi attack . . . 27

2.4.2 The QSEE privilege escalation vulnerability . . . 29

2.5 Requirements . . . 30

2.6 Introduction of capability models and CHERI . . . 32

2.6.1 Historical capability machines . . . 32

2.6.2 Overview of Capability Hardware Enhanced RISC Instructions, CHERI . . . 33

2.7 Summary . . . 40

3 A 64-bit compressed capability scheme for embedded systems **41**

3.1 CHERI for a 32-bit machine . . . 41
3.2 Existing capability encodings . . . 42
3.2.1 CHERI-256 . . . 43
3.2.2 M-Machine . . . 43
3.2.3 Low-fat . . . 45
3.2.4 Project Aries . . . 47
3.2.5 CHERI-128 Candidate 1 . . . 48
3.2.6 Summary . . . 48
3.3 The CHERI Concentrate (CC) and 64-bit capability encoding . . . 49
3.3.1 Balancing between precision and memory fragmentation . . . 49
3.3.2 Improved encoding efficiency . . . 55
3.3.3 The CHERI semantics . . . 58
3.3.4 CC-64 . . . 61
3.4 Summary of compressed capability schemes . . . 62
3.5 A CC-64 hardware implementation . . . 63
3.5.1 The MPU model . . . 63
3.5.2 Capability coprocessor vs. MPU . . . 65
3.5.3 Other considerations . . . 67
3.6 Summary . . . 67

4 CheriRTOS **69**

4.1 A CHERI-based RTOS . . . 69
4.2 Real-time operating systems . . . 70
4.3 Prerequisites . . . 72
4.3.1 The MIPS-n32 ABI . . . 72
4.3.2 Enabling CHERI-64 in Clang/LLVM . . . 73
4.3.3 A baseline RTOS . . . 73
4.4 CheriRTOS . . . 74
4.4.1 Overview . . . 75
4.4.2 OType space . . . 76
4.4.3 Non-PIC dynamic task loading . . . 77
4.4.4 Context switch . . . 77

4.4.5 CCallFast 79
4.4.6 Return and real-time guarantees 83
4.4.7 Secure centralised heap management 85
4.5 Evaluation 87
4.5.1 The MiBench benchmark suite 87
4.5.2 Non-PIC and compartmentalisation 88
4.5.3 Fast and direct domain crossing 89
4.5.4 Register safety, return and real-time guarantees 91
4.5.5 Overall system performance 92
4.6 CheriRTOS vs. state-of-the-art 94
4.7 Summary 95
4.7.1 The problem of temporal memory safety 96

5 Temporal safety under the CHERI architecture **97**
5.1 Background 97
5.2 Opportunities of CHERI temporal safety 100
5.2.1 Deterministic temporal memory safety 100
5.2.2 Spatial and temporal safety combined 101
5.2.3 Possible but inefficient 102
5.3 Optimising for efficient sweeping revocation 102
5.4 Architectural/microarchitectural proposals and implementations for fast sweeping 103
5.4.1 CLoadTags 103
5.4.2 Page table *cap-dirty* bit 105
5.4.3 Core dump study 106
5.4.4 Performance of fast sweeping 108
5.4.5 Pointer concentration 109
5.5 New allocator design for reduced sweeping frequency 110
5.5.1 Brief overview of dlmalloc() 110
5.5.2 Implementation of dlmalloc_nonreuse() 110
5.5.3 Experimental setup 114
5.5.4 Overall overheads of dlmalloc_nonreuse() 117

5.5.5 Breakdown of overheads . . . 118
5.5.6 Tradeoff between space and time . . . 121
5.6 Alternative sweeping schemes and concurrency . . . 122
5.6.1 Subset testing revocation . . . 122
5.6.2 Concurrent revocation . . . 123
5.7 Summary . . . 123

6 Conclusion **125**

6.1 Contributions . . . 125
6.1.1 A capability format for 32-bit cores . . . 126
6.1.2 CheriRTOS . . . 126
6.1.3 CHERI temporal memory safety . . . 127
6.2 Future work . . . 128
6.2.1 CHERI-64 . . . 128
6.2.2 CheriRTOS . . . 128
6.2.3 CHERI temporal memory safety . . . 128
6.2.4 Adopting capability protection in future embedded devices . . . 129
6.2.5 Extrapolating to non-CHERI systems . . . 130
6.2.6 Adversarial security evaluation . . . 130

A CHERI Concentrate bounds and region arithmetic **131**

A.1 Encoding the bounds . . . 131
A.2 Decoding the bounds . . . 133
A.3 Fast representable limit checking . . . 136

References **139**

List of Figures

2.1 Memory accesses controlled by MPU 25
2.2 Escalations towards a successful control flow hijack. **size** (brown) and **pointer to next/unused** (green) are allocator metadata fields. **data** (blue) is the actual allocated memory. Red indicates contaminated fields. . 27
2.3 The capability coprocessor 36
2.4 256-bit memory representation of a capability 38
2.5 Memory hierarchy and the tag cache 39

3.1 M-Machine capability encoding with example values 43
3.2 Low-fat capability encoding with example values 45
3.3 Low-fat bounds decompression 46
3.4 Aries capability encoding 48
3.5 Internal vs. external fragmentation 50
3.6 Heap internal fragmentation 53
3.7 Percentage increase in peak size of total stack allocations (SPEC CPU 2006 experimental builds) 53
3.8 Improved Low-fat Encoding with Embedded Exponent and Implied T_8 . . 57
3.9 The percentage of allocations that cannot be precisely represented in a capability. Lower is better. 58
3.10 CHERI Concentrate bounds in an address space. Addresses increase upwards. The example shows a `0x600`-byte object based at `0x1F00`. 60
3.11 64-bit CHERI Concentrate 62

4.1 Overall structure 75
4.2 Example of memory access instructions under CHERI. "`$`" denotes registers. Loading a word at address `$s0 + 8` (relative to the base of the capability) into `$t0`, either implicitly or via an explicit capability register. 76
4.3 Increment two variables at 0x800 and 0x900. Binary compiled for 0x0 now loaded at 0x11000. Red indicates patching. 78
4.4 CCallFast sequence 79
4.5 Trusted stack and CCallFast round trip. Dark indicates kernel-only objects. The pointer field of $KR1C points to the top of the trusted stack. . 83

4.6 Interrupt routine to check for expired CCalls 84
4.7 Memory allocator structure (gray boxes indicate that the bucket ID is sealed inside a sealed capability) 86
4.8 Instruction and cycle counts for a round trip: direct jump vs. capability jump (fast CCall) vs. exception based CCalls 90
4.9 Overhead of different protection levels 92
4.10 Overall overhead across benchmarks 93

5.1 Tags in the CHERI memory hierarchy and the refactoring of caches 103
5.2 CLoadTags. Assuming 128-byte cache lines and 64-bit capabilities. 104
5.3 **c** and **p** represent **cdirty** and **pdrity** bits. At the end Ck0 and Ck2 are unlinked from the queue and the newly freed chunk (in the middle) will be coalesced with Ck0 and Ck2 into Ck3 and inserted at the tail of the queue. After a revocation, Ck3 is returned to the free lists to be reused. . . 114
5.4 Overheads compared with results reported by other state-of-the-art techniques. CHERIvoke represents the new dlmalloc. 117
5.5 Run-time overhead decomposition for the constituent parts, with the default 25% heap overhead. 118
5.6 Memory bandwidth achieved for the sweep loop with different optimisations. The system's full bandwidth is 19405MiB/s. 120
5.7 Normalised execution time for the two workloads with highest overheads, at varying heap overhead. Default setup shown by dotted line. 121

A.1 CHERI Concentrate bounds in an address space. Addresses increase upwards. To the left are example values for a `0x600`-byte object based at `0x1E00`. 134

List of Tables

2.1 Example CHERI instruction-set extensions 34

3.1 Comparison of capability encodings 63
3.2 FPGA resource utilization and timing 65

4.1 MiBench: non-PIC vs. PIC (all numbers in billions) 89
4.2 Spectrum of protection levels and overhead 91

5.1 Sweeping performance of sample applications. The percentages in the last four columns indicate relative **reduction** compared with the baseline. Negative means **overhead** instead of reduction. Tildes indicate negligible numbers. 108

Chapter 1

Introduction

Today, embedded systems are deployed ubiquitously among various sectors, including automotive, medical, robotics and avionics. As these systems become increasingly connected, their attack surfaces increase dramatically to a much more sophisticated level, invalidating previous assumptions that such devices are susceptible only to direct physical attack. Also, the end of Moore's Law for single core and the continuing increase in transistor count drive manufacturers to pair large performance cores with smaller dedicated embedded cores for I/O and other peripherals on System-on-Chips (SoCs). As a result, securing the system without securing the embedded side of the chip still exposes attack vectors. For example, the Broadcom WiFi stack exploit enables the attacker to hijack the control flow via the Wi-Fi chip in nearly all iPhones and many Android phones [5]; the CAN bus in self-driving cars can be updated with malicious firmware to remotely control its motion by an attacker driving in parallel [14]. Even though many published vulnerabilities are well-known and well understood, it remains a challenge to build a comprehensive memory security framework around deeply embedded processors. Such systems with very small CPU cores typically impose tight constraints on the extra hardware logic, memory overhead and additional latency of any protection mechanisms deployed. Moreover, the lack of virtual memory only exacerbates the problem of memory protection, since all tasks reside within a single physical address space without isolation.

In this thesis, I work on many fronts of Capability Hardware Enhanced RISC Instructions (CHERI), a capability architecture developed by the University of Cambridge and SRI International. The CHERI Instruction Set Architecture (ISA) provides

direct processor support for fine-grained memory protection and scalable compartmentalisation on top of paged virtual memory. More importantly, CHERI hardware and software support for fast domain crossing has been proven to be much more efficient than conventional process-based isolation [67]. I hypothesise that the CHERI platform provides a novel angle towards low-overhead and scalable memory security for embedded devices, since the fine-grained memory protection, the scalable domain isolation and fast domain crossing within a flat address space are the desired properties in an embedded CPU, none of which is achievable via state-of-the-art solutions. However, challenges remain since existing CHERI implementations focus on a UNIX-based operating system (FreeBSD) on 64-bit processors and optimise for performance, whereas embedded processors are typically 32-bit, operate on bare-metal or on a Real-Time Operating System (RTOS) and optimise for low latency and real-time guarantees. Therefore, my research investigates how CHERI capabilities can be adapted to 32-bit embedded CPUs without violating the real-time constraints, and how a capability-based RTOS offers fine-grained memory protection and efficient, scalable task isolation/communication in a physical address space.

Further, I look into the possibility of practical temporal memory safety (e.g., against use-after-free attacks) under CHERI. I hypothesise that the separation between pointers and data and capability unforgeability fundamentally guarantee full temporal safety as opposed to probabilistic defenses on conventional architectures. Even so, the high cost of frequently sweeping memory to invalidate stale capabilities means that currently CHERI temporal safety is only possible, but not feasible. To address this, I explore hardware optimisations and new memory allocator designs to significantly reduce the cycle overhead and DRAM traffic for capability sweeping revocation, making it practical for most applications to adopt CHERI temporal memory safety.

1.1 Contributions

- Contribution to various capability compression schemes especially the CHERI Concentrate (CC) format. CC-128 compresses 256-bit capabilities into 128 bits. It halves the memory footprint of capabilities, improves encoding efficiency, maintains compatibility with legacy C code and addresses pipelining issues.

- Restructuring BERI with 32-bit addressing and implementing CHERI-64, a compressed 64-bit capability scheme and a 64-bit capability coprocessor for the 32-bit CPU.
- Various fixes and patches to upstream LLVM to enable 32-bit addressing for MIPS in the compiler. Adding support for CHERI-64 on top of the MIPS 32-bit compiler in Clang/LLVM.
- A real-time kernel, CheriRTOS, which enforces fine-grained memory protection and efficient task isolation via capabilities.
- CHERI temporal memory safety study. I introduce new instructions and micro-architectural optimisations for fast memory sweeping, and implement a non-reuse memory allocator for reduced memory sweeping.

1.2 Publications

- Xia, H., Woodruff, J., Ainsworth, S., Filardo, N. W., Roe, M., Richardson, A., Rugg, P., Neumann, P. G., Moore, S. W., Watson, R. N. M. and Jones, T. M. ‘CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety’. In: *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: ACM, 2019, pp. 545–557.
- Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R., Chisnall, D., Davis, B., Gudka, K., Filardo, N. W., Markettos, A. T., Roe, M., Neumann, P. G., Watson, R. N. M. and Moore, S. W. ‘CHERI Concentrate: Practical Compressed Capabilities’. In: *IEEE Transactions on Computers* (2019)
- Xia, H., Woodruff, J., Barral, H., Esswood, L., Joannou, A., Kovacsics, R., Chisnall, D., Roe, M., Davis, B., Napierala, E., Baldwin, J., Gudka, K., Neumann, P. G., Richardson, A., Moore, S. W. and Watson, R. N. M. ‘CheriRTOS: A Capability Model for Embedded Devices’. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018, pp. 92–99
- Joannou, A., Woodruff, J., Kovacsics, R., Moore, S. W., Bradbury, A., Xia, H., Watson, R. N. M., Chisnall, D., Roe, M., Davis, B., Napierala, E., Baldwin, J., Gudka, K., Neumann, P. G., Mazzinghi, A., Richardson, A., Son, S. and Markettos, A. T. ‘Efficient Tagged Memory’. In: *2017 IEEE International Conference on Computer Design (ICCD)*. Nov. 2017, pp. 641–648

1.3 Dissertation overview

In Chapter 2, I present a short survey of the current processors used in embedded devices. Then, I present the growing problem of security and memory safety in such systems, accompanied by case studies on the typical attack vectors. The next section introduces and describes the state-of-the-art memory security schemes and techniques, followed by comparison and analysis on the effectiveness and shortcomings of said approaches. From the analysis I extract the fundamental requirements of memory safety for embedded systems.

Chapter 3 describes the work I have done to investigate a new compressed capability format for embedded processors. The study consists of the design and implementation of the 64-bit CHERI Concentrate encoding. I evaluate the memory overhead due to low precision and the hardware implementation complexity compared with other state-of-the-art security components, drawing the conclusion that a 64-bit compressed capability machine for 32-bit embedded devices is feasible.

Chapter 4 presents my work on a proof-of-concept RTOS kernel, CheriRTOS, using CHERI as the only memory protection and isolation mechanism atop CHERI-64. The evaluation shows that a capability-aware RTOS can be implemented without violating the constraints of typical embedded systems.

Chapter 5 visits the topic of enforcing temporal memory safety under the CHERI architecture. I propose, investigate and implement new ISA and micro-architectural changes to significantly accelerate sweeping revocation for capabilities. Further, I implement a non-reuse version of the `dlmalloc` memory allocator which avoids the reuse of memory allocations and reduces the rate required for sweeping. Combined, the proposed changes bring the cost of sweeping revocation down by orders of magnitude, making it feasible to apply CHERI temporal safety to many applications.

Chapter 6 draws conclusions.

Chapter 2

Background

One major focus of this thesis is bringing capability-based protection to the 32-bit embedded space. I begin by defining what it means to operate in the embedded space. Next, I describe the situation of memory safety for such systems, presenting literature review and case studies to illustrate the status quo of memory protection for embedded devices. Based on the advantages and shortcomings of state-of-the-art memory safety schemes, I identify and summarise the requirements of a secure design. Finally, I present background knowledge of CHERI.

2.1 Introduction to embedded systems

Embedded systems and processors are deployed among various sectors, which typically include devices ranging from tiny chips like in video cables, keyboard controllers and sensors, to small ones like Wi-Fi and security chips in mobile phones, to even larger systems like routers. They differ vastly from normal desktop and server systems in that they do not perform general purpose computing, instead they are heavily adapted to domain-specific areas and custom designs, performing only a handful of dedicated tasks. Moreover, software running on top of it does not commonly involve general purpose operating systems with a full stack of capabilities (e.g., the abstraction of files, processes, networking, etc.). Instead, code either runs on bare-metal (no OS) or under a Real-Time Operating System (RTOS), providing limited abstractions and interfaces for dedicated computations. In terms of computing instances, an RTOS

creates and runs lightweight *tasks* in a shared flat address space, an analogous but simpler model to processes in UNIX operating systems.

Another major difference are the primary requirements. A PostgreSQL database server running under Linux, for example, focuses on high processing power, high throughput and multi-processing to resolve the large number of requests from clients. On the contrary, the goal of embedded applications is typically low-latency, determinism and deadline guarantees. For example, a self-driving vehicle has to respond promptly in the steering and brake system when a danger arises. The system ensures that it responds to real-time events within a pre-determined delay. Failing to guarantee real-timeness and determinism may put the user into critical danger, whereas the raw performance of such processors may not be of primary concern. In addition, these devices are commonly used in highly constrained scenarios including low-cost and low-power applications. As a result, they often come with a limited amount of RAM and ROM and are not clocked at high frequencies, and commonly have a very simple cache hierarchy or no caches at all. This leads to many important design decisions like minimising code size, the support for low-power programming and deep sleep modes.

Although the above describes the typical characteristics of embedded systems, many devices today are labelled “embedded”, covering a wide range of different specifications. From the very low end, we have a keyboard controller that is an 8-bit processor operating at only a couple of megahertz, to an Amazon Fire TV stick with a quad-core Snapdragon 8064 and 2GiB RAM, and an Android-based operating system. As the scope may be too broad and some “embedded systems” already deviate a lot from the requirements above, I would like to narrow down the definition so that the target of this thesis is unambiguous.

The specifications of the embedded systems discussed throughout this thesis should be:

- Used in low-cost and low-power scenarios.
- Limited memory (typically under 1MiB), very simple or no cache hierarchy.
- Short in-order processor pipeline.

- Flat physical address space. No address translation, virtual memory or Memory Management Unit (MMU).
- Code runs on bare-metal or an RTOS.
- Focuses on determinism and real-time constraints in addition to performance.
- Priority-based or deadline-driven task scheduling for multitasking.
- Connected to other systems, either wired or wireless, i.e., direct physical attacks are not the only possible vulnerability.

2.2 The need for memory safety

The rapid growth of the market of embedded systems and the increase of connections across devices pose new threats and challenges. Today, even the smallest embedded products (e.g., smart watches and earphones) are connected to various other devices via connections like Wi-Fi or bluetooth. The increased connectivity quickly invalidates our previous assumption that these devices are susceptible primarily to physical attacks, and exposes a whole new variety of attack surfaces.

Recently, new attack vectors targeting memory safety have been disclosed, either exposing a crucial vulnerability for a specific device, or in terms of systematic surveys of attack surfaces. For instance, the Broadcom Wi-Fi vulnerability disclosed in 2017 [5] allows arbitrary code execution in the Wi-Fi chip of many mobile phones. Costin, A. et al. conduct a large-scale survey of a total of 32,356 firmware images of common embedded devices, extracting RSA keys, password hashes as well as identifying backdoors. The study concludes that an even broader analysis is necessary to systematically understand the vulnerabilities in embedded firmware [17]. These vulnerabilities are only a tip of the iceberg of all the possible attacks discovered in recent years, and with the rapid growth of Internet of Things (IoTs), we can only be prepared for further disclosure of even more sophisticated attacks. In fact, papers and articles from academia and industry (e.g., [65, 42, 24]) have already warned us of the difficult situation and other challenges of enforcing security in embedded and IoT devices. With more than 200 billion IoT devices in total to be shipped by 2021 [24], and several security platforms only starting to emerge now, we do not anticipate the security aspect to be effectively addressed in the near future.

2.3 State-of-the-art memory protection

2.3.1 Desktop systems

Memory corruption bugs and attacks have been well studied in past decades with countless projects and papers on multiple fronts. In summary, we are able to categorise these bugs into two major categories, that is, spatial or temporal memory safety violations. The paper *Eternal War on Memory* offers a good summary and overview of a wide range of memory bugs, attacks and potential mitigations [62], which describes how such small bugs can be leveraged and quickly escalate into severe system vulnerabilities, including code injection, control flow hijack, data attacks and information leak. Besides the exposure of memory corruptions, defense and mitigation techniques have been proposed and implemented to counteract the damage, which include both hardware and software approaches, for example: virtual memory for address space isolation and page-level protection, AddressSanitizer [56] to detect out-of-bounds, use-after-free, use-after-return access, memory tagging (SPARC Silicon Secure Memory (SSM) [50] and AArch64 Hardware ASan (HWASAN)) for temporal safety violation detection, StackGuard [18] to guard against stack overflows by inserting canaries, Control Flow Integrity (CFI) schemes [12] to defend against Jump-Oriented-Programming (JOP) and Return-Oriented-Programming (ROP) attacks, fat-pointers [46, 49] to bounds check on a per-pointer level, and so forth. Many of the defense techniques have already been used in production with reasonable strength and performance.

2.3.2 Searching for novel solutions for embedded systems

Unfortunately, memory safety for desktop computers mentioned above can exhibit high overhead. AddressSanitizer, for example, incurs more than 2× slowdown and memory overhead for memory intensive workloads [56]. To make matters worse, determinism is a common requirement for low-latency use cases which is overlooked in the aforementioned studies. MMUs, for example, introduce non-determinism due to Translation Lookaside Buffer (TLB) misses and page table walks, even though the TLB hardware is not too expensive to incorporate. Therefore, virtual memory is typically absent from embedded systems.

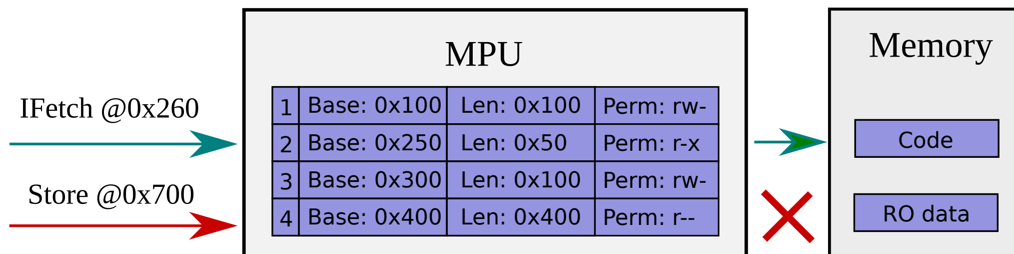


Figure 2.1: *Memory accesses controlled by MPU*

These limitations mean well-studied solutions rarely scale down, and novel solutions are developed and specifically tailored to these devices. The Memory Protection Unit (MPU) is commonly adopted in embedded processors to mark memory regions with security attributes [2] to prevent arbitrary physical memory access (Figure 2.1). The intention is to protect crucial code and data from buggy or malicious user space tasks. Although widely used in industry, MPUs have several inherent drawbacks. First, an MPU is implemented as a kernel space device and each register takes several cycles to configure, and some memory mapped implementations have even higher latencies through the memory hierarchy. As a result, they are normally configured only globally at system start-up, which makes per-task memory access control difficult, and user space cannot adopt it for intra-task protection. Second, the number of MPU entries is limited. With only 8 MPU regions in most implementations [2], only security critical memory partitions are protected, e.g., the kernel, encryption keys, code sections, etc., thus any fine-grained memory protection with MPUs is a challenge. Third, MPU lookups involve associative searches of all entries. Each cycle can potentially require up to 32 (accounting for both instruction fetch and data) comparisons with 8 MPU entries. This means that MPUs could inherently be inefficient in terms of power and die area. To compare CHERI with the MPU model in this thesis, I implement the RISC-V Physical Memory Protection (PMP) unit, which is a state-of-the art MPU component. The RISC-V PMP has an open-source specification [66] that can be followed easily.

In addition to MPUs which generally provide system-wide memory protection for critical regions, TrustZone® partitions memory into secure and non-secure worlds [64] with constrained control flow. Non-secure code can only jump to valid entry points on the secure side, and secure code calls non-secure functions after clearing registers and pushing the return address on the secure stack to prevent data leak and to protect

the return address. Nevertheless, TrustZone is likely to encounter scalability issues. For example, further isolation within a world is not possible, still enabling attacks in the same world. A task from one user is visible (and potentially modifiable) from another, and buggy drivers in the secure world can have access to secrets and secure keys which are stored in the secure space as well.

Besides commercial implementations, research projects also tackle the problem of embedded system memory safety. TrustLite [37] and TyTAN [9] extend the MPU and kernel to provide data isolation, trusted inter-task communication, secure peripherals, run-time configurability, etc. The Execution-Aware MPU (EA-MPU) links data and code entries and tags them with task identifiers. Therefore, only the entries of the active task are enabled, providing per-task protection. However, an MPU-based approach inherits several flaws described above, the most significant being that the number of protected regions is still severely limited by the number of MPU entries. If the kernel attempts to support more tasks and regions than MPU entries, then expensive system calls have to be made to swap MPU registers when the desired one is not present. TrustLite argues that this bottleneck may not be a problem as the number of simultaneous tasks in embedded systems is low. However, as the market rapidly grows, I would like to see scalable and future-proof solutions in case this assumption cannot be safely made in the near future.

For control flow robustness and defense against Return-Oriented Programming (ROP) attacks, architectures have been developed for embedded systems [20, 21] by using dedicated instructions for function calls, exposing only valid entry points, hiding return addresses in protected spaces and so forth in a similar approach to TrustZone. On the other hand, Sanctus [48, 61] builds tasks into Self-Protecting Modules (SPMs) to restrict access and enforce control flow with a minimum Trust Computing Base (TCB). However, it sacrifices software flexibility and incurs a high hardware cost by implementing SPM loading, measurement and runtime identification in the trusted CPU.

We have also seen research projects tackle the memory protection problem on a programming language level for embedded devices. *nesCheck* [45] modifies the language and compiler to perform stronger type safety, type validation, static analysis and run-time checks, carrying additional metadata to remove or detect unsafe memory access. Others develop new compiler frameworks to address stack, array and pointer

safety, and implement new heap allocation techniques to address heap safety without introducing garbage collection [25]. These language- and compiler- based schemes have shown good results within reasonable overhead and without substantial hardware changes. However, a software approach protects code written or compiled under specific language variants or toolchain. If a task is compromised or is itself malicious, it may directly execute or inject low-level code that circumvents any language level invariants. Similar to other software schemes for desktop systems, this type of memory protection is most useful for debugging existing codebase rather than offering system-wide security guarantees.

2.4 Case studies

2.4.1 The Broadcom Wi-Fi attack

The Broadcom's Wi-Fi chipset is widely deployed in smartphones including the Nexus series, Samsung products and all iPhones since iPhone4. A successful exploit was recently discovered and published in such a system [5]. The Wi-Fi chipset, BCM4339, is a Cortex-R4 based SoC running firmware from a 640KiB ROM and processing data from a 768KiB RAM.

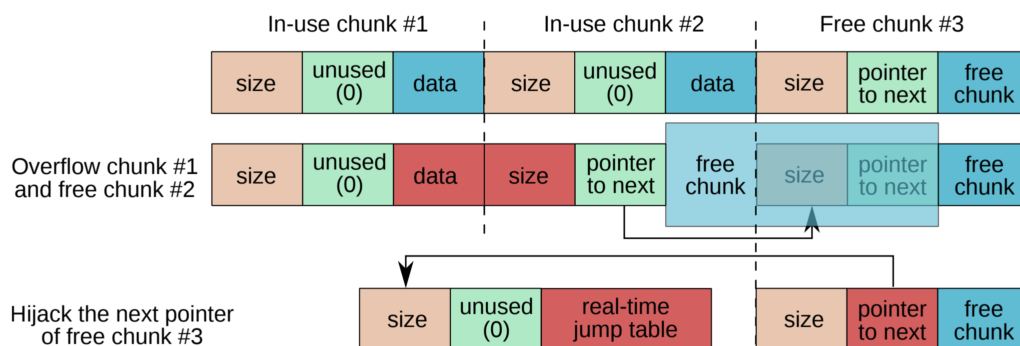


Figure 2.2: *Escalations towards a successful control flow hijack.* ***size*** *(brown) and* ***pointer to next/unused*** *(green) are allocator metadata fields.* ***data*** *(blue) is the actual allocated memory. Red indicates contaminated fields.*

Figure 2.2 illustrates how a slight buffer overflow is exploited which escalates to a complete control flow hijack. To summarise:

1. Reverse engineering reveals a slight overflow in a heap allocation (chunk #1). which is sufficiently large to overwrite the size field of an adjacent in-use allocation (#2).
2. Chunk #2 is deallocated when the Wi-Fi connection closes. It is linked into the free-list before chunk #3 with a hijacked size field.
3. The attacker carefully crafts a request that fits into the hijacked size and is allocated to chunk #2, which now overlaps with #3.
4. The *next-pointer* field of #3 is under the control of the attacker. It is overwritten with a value that points to a kernel allocation storing the real-time jump table. Now the in-use allocation containing the kernel jump table is exposed in the free-list.
5. The attacker sends another crafted request that gets allocated to the jump-table memory chunk and overwrites them with pointers to malicious code.
6. A real-time event fires. The kernel fetches the hijacked pointer in the jump table and jumps to malicious code.

Again, this is another typical example of a minor buffer overflow leading to arbitrary code execution. After dissecting the firmware, the reverse-engineered code shows that the programmer actually has memory safety in mind. For example:

```
1 // Copying the RSN IE
2 uint8_t* rsn_ie = bcm_parse_tlvs(..., 48);
3 if(rsn_ie[1] + 2 + (pos - buffer) > 0xFF) {
4     ... // Handle overflow
5 }
6 memcpy(pos, rsn_ie, rsn_ie[1] + 2); pos += rsn_ie[1] + 2;
```

The code checks for an overflow before copying data. Both the difficulty and the run-time overhead forbid the programmer to add manual bounds checks to every

potential overflow, leaving several of them still hidden and exploitable. One may argue that the proper use of MPU could prevent such attacks, and in fact the Cortex-R4 CPU of the Wi-Fi chip implements it. However, we also imagine the difficulty of adapting it to this use case. Heap allocations require fine-grained protection on every memory chunk, which unfortunately cannot be achieved with the coarse-grained MPU with limited regions. At best, a programmer could segregate the heap into a kernel and a user pool and dedicate two MPU entries, which still does not stop memory attacks between users or between the kernel and buggy drivers in the kernel space.

The problem of centralised heap management. The Wi-Fi attack shows a fundamental problem of memory allocation. In typical embedded systems with a small flat physical address space, dynamic memory allocations are commonly done in a centralised heap for all user tasks (or even including the kernel). Although many dynamic heap allocators actually have low and relatively deterministic latencies and are widely used among multiple systems including mbedOS, FreeRTOS [4] and TinyOS [45], the security issues caused by dynamically allocated and shared memory pool have never been properly addressed. Without bounded access, allocations from a user can easily overflow into other allocations or even the heap metadata to attack other users or the heap allocator itself. To hide and protect heap metadata, many allocators including jemalloc [28] used in FreeBSD organise the metadata in separate data structures. However, embedded systems still prefer to attach metadata directly beside each allocated chunk and use "free-lists" like dlmalloc [40] in mbedOS for lower latency. The organisation of such allocators demands low-latency fine-grained memory protection for effective isolation, which unfortunately is difficult to achieve with current software-hardware platforms, and inherently creates the tension between flexibility and security. Many designs simply revert to pre-determined static allocations for each task to avoid this problem, losing all flexibility of a dynamic heap.

2.4.2 The QSEE privilege escalation vulnerability

The Qualcomm's Secure Execution Environment (QSEE) uses TrustZone technology to contain trusted applications (trustlets) in the secure world beside a normal embedded OS. Trustlets perform privileged operations like key storage and Digital Rights Management (DRM). Only few applications in the normal world are able to load and communicate with the trustlets in a controlled manner, and the MPU is set to protect the regions of trustlets against data attacks and code injection. Recently,

an attack has been found to exploit one of the trustlets to execute arbitrary code in the secure domain [6]. The attack starts in a similar way to the Broadcom WiFi attack by uncovering a crucial buffer overflow. The details are more complex and will not be listed here.

The author of [6] observes that all trustlets reside in the secure world and TrustZone only offers a uni-directional trust relationship. Two key properties arise from this observation. One, a successful exploit on a single trustlet can expose all other trustlets in the secure world. Two, the secure world has strictly higher privileges, which is able to read, write and execute any memory in the normal world as well. In this example, part of the gadgets is simply injected to the normal world and is executed by the indirected trustlet, which in turn attacks the embedded OS and other parts of the system.

In addition to memory protection, this attack exposes the insufficient level of isolation and scalability provided by TrustZone. Having only two worlds forces all trustlets to live in a single domain, exposing them to possible attacks from one another. Also, TrustZone employs a traditional hierarchical and uni-directional approach which fails to guarantee mutual distrust. In the attack, it is questionable to run the embedded OS in the normal world while the trustlets live in the secure world. This design suggests that trustlets have strictly higher privileges than the OS, which is not true. The embedded OS can be attacked from the secure side even though it contains no vulnerabilities itself. A desirable design should allow mutual distrust domains, where they are parallel instead of being a superset or subset of each other.

2.5 Requirements

Although the case studies are just two data points, most security vulnerabilities originate from similar memory safety issues. In fact, Microsoft security engineers have confirmed that around 70% of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues [63]. From the common attack surface in security vulnerabilities, the survey of state-of-the-art solutions, the shortcomings of existing protection schemes and the properties and growth of embedded systems themselves, I identify the following requirements that are essential to a secure design.

Task isolation. In a flat physical address space, it is essential to separate multiple tasks from different users or vendors into different domains. Unconstrained access means a malicious task can easily compromise other critical components in the system. In addition, I would like to include mutual distrust as part of the definition, as the case study demonstrates that a uni-directional trust model is insufficient.

Fine-grained memory protection. So far, memory protection in embedded systems controls access to large segments of code and data. Protecting finer granularities often require explicit checks from the compiler or programmer at a cost of run-time slowdown [25, 45], and can be error-prone or incomplete [5]. An architecture should provide a generic mechanism for low-cost and fine-grained memory protection.

Fast and secure domain crossing and inter-task communication. A system with multitasking often requires communication among components, either in terms of message passing and memory sharing, or in cross-domain function calls. Strong task isolation should not prohibit efficient and secure domain crossing and communication.

Secure centralised heap management. As discussed in Section 2.4.1, existing protection schemes often fail to guarantee that memory management in embedded systems is both flexible and secure. A system with fine-grained memory protection enables a secure shared-heap allocator by restricting any user of allocations from reaching metadata or any other allocations.

Real-time guarantees. No security architecture should violate real-time constraints. Cached memory translation (like virtual memory with MMU), for example, directly violate the low-latency and deterministic properties of embedded systems and therefore should not be used for protection.

Scalability. We have seen the rapid growth of the embedded market in recent years and a scalable solution is desirable. MPU-based approaches described above suffer from scalability issues and may not meet the security needs as these systems become more capable and dynamic.

A generic solution for security. Embedded chips often implement multiple components to enforce safety. ARM solutions typically provide an MPU, a Security Attribution Unit (SAU), TrustZone® and even an Implementation Defined Attribu-

tion Unit (IDAU) for security. Together, these components are able to defend against common attacks if configured correctly. However, orchestrating many security mechanisms is non-trivial, and we have seen in practice that vendors often revert to manual assertions and explicit checks, leaving these security layers largely unused. An ideal solution should be generic enough to ease the effort of orchestration and deployment.

2.6 Introduction of capability models and CHERI

2.6.1 Historical capability machines

In 1966, Dennis and Van Horn formally described a memory protection framework in which each process possessed a list of keys (capabilities) that granted access to objects within the system [23]. A capability is an unforgeable token, which when presented can be taken as incontestable proof that the presenter is authorized to have access to the object named in the token [54]. Capability systems are one of the architectural protection models among segmented memory, MMU protection, access control lists, etc. Capability systems have been implemented on various machines including IBM System/38, which created a unique ID for each capability and used a tagged architecture to avoid malicious capability manipulation [41], and the Cambridge CAP computer, which had a centralised capability table that also facilitated secure procedure calls [47]. Other than a hardware approach, a software capability system, Hydra, implemented an object-based OS with internal bit vectors for permissions, in which a set bit indicated a granted permission. *AND only* operations on bit vectors ensured that only a subset of permissions could be derived from given capabilities [76]. However, capability schemes in that era were not widely adopted due to the high overhead which at that time was not justified given the security provided, and have been overshadowed by coarse-grained yet low-overhead solutions like paged memory and MMUs.

Due to the rapid growth in available transistor resources and the increasingly sophisticated attacks on memory safety, capability machines have been revisited recently as a complementary or an alternative approach. Recent software capability models include Capsicum, which takes an incremental approach and serves as an extension for UNIX, providing new kernel primitives and userspace sandbox API via

capabilities [68]. The formally verified seL4 uses capabilities for memory management, authorisation, etc. [36]. To extend the Trusted Computing Base (TCB) to hardware and for a better performance, many architectural proposals have emerged, including the M-Machine [13] and the Low-Fat Pointer scheme [39], which provide hardware bounds checking and novel domain crossing mechanisms.

2.6.2 Overview of Capability Hardware Enhanced RISC Instructions, CHERI

CHERI, developed by SRI International and the University of Cambridge, revisited capability systems by extending the Bluespec Extensible RISC Implementation (BERI) platform and adding security extensions for the 64-bit MIPS Instruction Set Architecture (ISA) [69, 75]. The CHERI ISA provides direct processor support for fine-grained memory protection and scalable compartmentalisation on top of paged memory. Unlike previous capability implementations, CHERI maintains both a mixed model (traditional code mixed with capabilities) and the purecap-ABI (pure capabilities) to offer fine-grained protection while being backward compatible, therefore it is able to run a range of software including FreeBSD/CheriBSD and other user-space applications. BERI/CHERI is written in Bluespec System Verilog (BSV), a high-level functional HDL language, and the existence of other higher level implementations and simulation platforms (QEMU, L3) allow further extensions and rapid explorations.

CHERI capabilities

CHERI architecturally distinguishes capabilities from normal data. All memory accesses (including instruction fetch and data load and store) in CHERI must be done via capabilities. An access fails if the capability does not allow such an operation, either due to out of bounds or insufficient permission. To enforce integrity and unforgeability, each capability is tagged with a **tag** bit set, without which a capability is not valid to be used. In addition to bounded memory access with memory capabilities, sealed capabilities (**s** bit set) can be used for compartmentalisation and fast domain crossing. Sealed capabilities are created with another capability as a key. After the sealing operation, the capability is immutable and non-dereferenceable, and is given an Object Type (otype) from the key. CHERI restricts that capabilities can only be unsealed in two ways. First, the original key or another key with suffi-

Mnemonic	Description
CGetBase	Move base to a GPR
CGetLen	Move length to a GPR
CGetTag	Move tag bit to a GPR
CGetPerm	Move permissions to a GPR
CSetBounds	Set (reduce) bounds on a capability
CClearTag	Invalidate a capability register
CAndPerm	Restrict permissions
CLC	Load capability register
CSC	Store capability register
CL[BHWD][U]	Load byte, half-word, word or double via capability register, (zero-extend)
CS[BHWD]	Store byte, half-word, word or double via capability register
CToPtr	Generate DDC-based integer pointer from a capability
CFromPtr	CIncBase with support for NULL casts
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CJR	Jump capability register
CJALR	Jump and link capability register
CCall	Perform a capability call

Table 2.1: *Example CHERI instruction-set extensions*

cient rights can perform unsealing. Second, the Capability Call (CCall) mechanism takes a pair of sealed code and data capabilities with matching otypes and sufficient permissions. Upon a successful CCall, both the code and the data capability are unsealed and installed, transferring control to the new domain with the callee's code and data. CCall is the fundamental method to perform an atomic domain crossing across compartments under CHERI.

The CHERI ISA

The current CHERI ISA extends MIPS with capability instructions. It includes capability inspection (`CGetBase, CGetTag...`), manipulation (`CSetBounds, CClearTag, CAndPerm`), control flow (`CBTU, CBTS, CJR, CJALR, CCall...`) and memory instructions (`CLC, CSC, CL[BHWD]...`).

The example instructions (Table 2.1) demonstrate the important monotonicity property of CHERI, that is, the ISA only permits capability operations that do not increase rights. Any capability manipulations only decrease bounds, reduce permissions or clear tags but never vice versa. Also, storing non-capability data to a location containing a capability will invalidate the tag to forbid overwriting an existing one to change its interpretation, which combined with monotonicity guarantees unforgeability. Such restrictions ensure that a program is never able to fabricate arbitrary memory references; therefore, its ability to reference memory is strictly limited by the unforgeable capabilities it receives from the system, and a protection domain is defined by the transitive closure of memory capabilities reachable from its capability register set.

In address-space sandboxing and compartmentalisation: The fact that a domain is defined by its reachable capabilities facilitates CHERI-based compartmentalisation. The ISA defines a `CCall` operation that atomically switches the set of root capabilities (including code and data), transferring control and reachability to the callee. The `CCall` mechanism allows scalable construction and transition of compartments within a process. Compared with multi-process sandboxing that switches between processes and address spaces, `CCall` is orders of magnitude cheaper and achieves the same magnitude of overhead as direct function calls. For example, the CHERI-enabled FreeBSD OS presents at least three mechanisms to transfer data to another isolated domain: *pipe*, which transfers to a sandbox process via a UNIX pipe; *shmem*, which transfers to a sandbox process through shared memory using a semaphore for synchronisation; *CHERI domain crossing*, which transfers to a in-process sandbox via a CHERI capability and CCall. A round trip on a tiny dataset (caller prepares data; callee calls `memcpy` to copy from a shared buffer to its own domain, which for this dataset takes only 150 cycles in a plain function call) takes >17,000 cycles under multi-process isolation (`pipe`, `shmem`), whereas a CHERI call/return round trip gives a fixed overhead of around 250 cycles [67]. In this simple case, CHERI overhead comes from clearing registers as well as preparing and validating callee capabilities, unlike multi-process isolation which exhibits high overhead from system calls and OS synchronisation. For large datasets, the fixed overhead of 250 cycles from CHERI becomes negligible, showing the same magnitude of performance to plain function calls, whereas the cost of cache and TLB flushes still dominates the data transfer under multi-process isolation. The *pipe* case performs additional data copies and converges to about 6 times more expensive than function calls and CHERI.

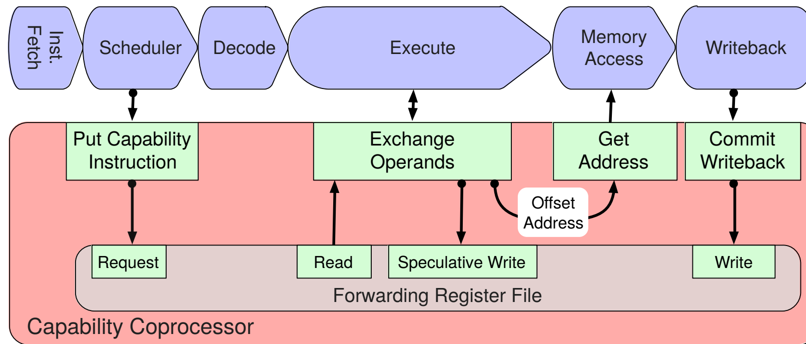


Figure 2.3: *The capability coprocessor*

CHERI compartmentalisation enables intra-process isolation, which avoids the costs incurred from inter-process communication, like kernel synchronisation, data copying, cache flushes, TLB maintenance, etc..

Hardware implementation

The current CHERI processor is implemented in Bluespec SystemVerilog [7] as a modularised extension to the MIPS R4000 CPU. As with the MIPS R4000, our base processor (Bluespec Extensible RISC Implementation (BERI)) is single-issue and in-order, with a throughput approaching one instruction per cycle. BERI has a branch predictor and uses limited register renaming for robust forwarding in its 6-stage pipeline. BERI runs at 100MHz on an Altera Stratix IV FPGA and is capable of running the stock FreeBSD 10 operating system and associated applications.

The CHERI processor adds capability extensions to BERI and is fully backward compatible, facilitating side-by-side comparisons. CHERI capability extensions are implemented as a MIPS coprocessor, CP2. Similar to the MIPS floating-point coprocessor, CP1, the capability coprocessor holds a new register file and logic to access and update it. The MIPS pipeline feeds instructions into the capability coprocessor, exchange operands with it, and receive exceptions from it (Figure 2.3). The capability coprocessor also transforms and limits memory requests from instruction fetch and MIPS load and store instructions. [75]

Although one may choose to understand the capability coprocessor by comparing it to a normal MIPS coprocessor, the former is far more integrated with the main pipeline and memory hierarchy than the latter. For example, capability bounds and

permission checks have altered the exception behaviour of jumps and branches; several system registers like the Program Counter (PC) have been replaced by bounded capabilities which affect the execution of the main pipeline; the main CPU and all cache hierarchies require significant change to natively understand memory tags and capabilities. In summary, the capability coprocessor affects and cooperates with the main CPU in ways that a normal MIPS coprocessor cannot, and care is needed to make direct comparisons.

The current CHERI CPU implementation adds a separate register file for capabilities, similar to a MIPS floating point coprocessor holding floating point registers. A design space worth exploring is the choice between a *split* and *merged* register file. A merged register file extends existing general purpose registers in the main CPU to also hold capabilities. The two designs have been discussed extensively in the architecture document [70] and are being further investigated. The major distinctions include:

- The coprocessor interface. A coprocessor design holding a separate register file fits nicely into the existing CPU structure and is easily configurable as an extension, whereas a merged register file needs a deep restructure of the CPU pipeline.
- Context switches. The split design introduces 32 capability registers as well as extra capability system and control registers, at least doubling the context size and increasing the latency of context switches. Such a large context is typically unacceptable in embedded and latency-sensitive systems.
- Opcode space. Dedicated opcodes are required to perform capability operations in the capability register file. In contrast, a merged design can reuse the same opcodes for capability manipulations. Whether the opcode is interpreted as a capability instruction can depend on the mode of the CPU or whether the register has a valid capability tag.
- Intentionality. It is impossible to misuse a capability register as an integer or vice versa in the split design, since the instruction explicitly specifies the intended register file. With integers and capabilities merged, such confusions can happen, and explicit instructions with dedicated opcodes might still be required when necessary.

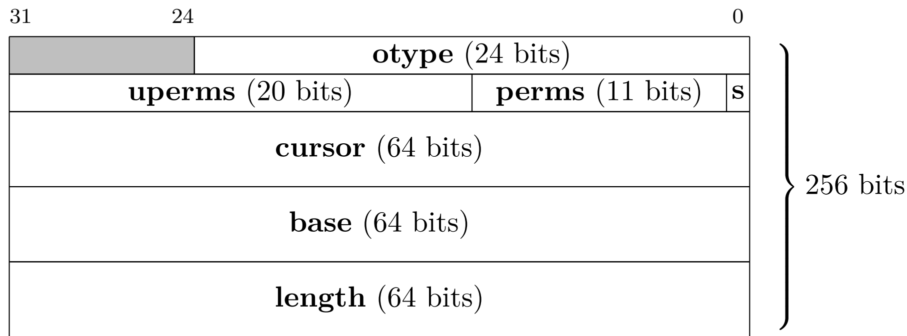


Figure 2.4: *256-bit memory representation of a capability*

- Compiler complexity. A merged register file can choose to extend all or part of the general purpose registers to accommodate capabilities. Since a capability doubles the size of an integer register, only allowing a subset for capabilities avoids doubling the register context size. This comes at a cost of additional compiler register allocation and ABI complexity. Therefore, this design is not being actively explored.

CHERI-256 encoding. Figure 2.4 shows the encoding of the 256-bit capability format. The format quadruples the pointer size by adding **otype**, **perms**, **base**, **length** fields. For memory capabilities, each dereference is checked against the bounds and permissions, and for sealed capabilities, they can only be unsealed by other capabilities that are able to unseal this **otype**. The **otype** field is also checked when performing a `CCall`. A `CCall` takes a pair of code and data capabilities, checking both **otype**s match before installing the unsealed capability pair of the new domain. **otype** matching guarantees that the new domain does not take an arbitrary sealed data capability.

The tag cache. To enforce unforgeability, the processor pipeline is modified so that each capability is extended with a tag bit in the register file and caches. However, it is much more difficult to manufacture a custom DRAM to accommodate tags. Instead, the CHERI processor uses existing off-the-shelf DRAM and partitions it into data and tags. Each memory request that reaches DRAM accesses data and its tag from the two partitions, which could potentially double the number of DRAM transactions.

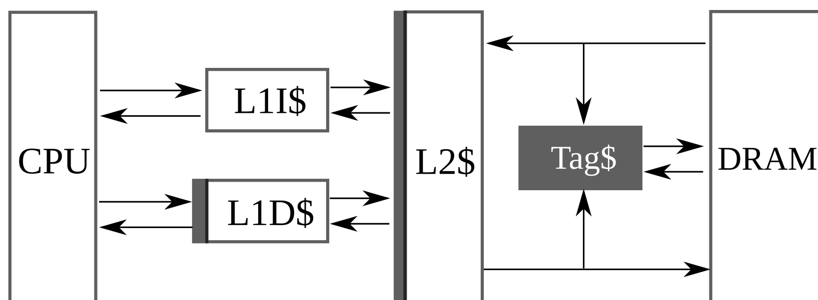


Figure 2.5: *Memory hierarchy and the tag cache*

The current implementation addresses this issue by adding a tag cache before the DRAM (Figure 2.5). The data request is sent to DRAM while its tag is looked up in the tag cache in parallel. Upon a hit, the tag response waits for the DRAM response before being combined and returned to the L2 cache. As the data transaction and the tag cache lookup happen in parallel, such an operation has the same latency of a single DRAM transaction.

Introducing a tag cache helps avoid another problem as well, which is the aliasing effect in DRAM. Without a tag cache, the two transactions for tags and data may conflict in DRAM if, for example, they occupy different rows in the same bank. For a small DRAM array which is capable of opening only a limited number of concurrent rows, such an effect further deteriorates performance. By introducing a tag cache, most tag operations will hit in the cache and will not require a second DRAM transaction, significantly alleviating the impact of DRAM aliasing.

Of course, the effectiveness depends on the tag cache hit rate, because a miss or an evict makes it no better than issuing two transactions to DRAM. Joannou, A. et al. proposes a design [34] that compresses tags in multiple hierarchies. The compression increases the effective capacity by exploiting the sparsity of tags in memory. By exploring the design space, the author chooses an optimised configuration that reduces the DRAM overhead by up to 99% for many non-pointer-heavy applications. For the aliasing effect, a DRAM overhead reduction of 99% means only 1 in 100 tagged memory transactions will actually issue two transactions to DRAM, while 99 hit the tag cache and therefore issue a single DRAM transaction. This greatly minimises the chance of aliasing even under limited row concurrency.

2.7 Summary

This chapter described the embedded systems that are in scope of this dissertation. I explained the different specifications and requirements between desktop and embedded systems and why this could lead to fundamental difficulties in enforcing memory safety for the latter. The state-of-the-art and literature review presented solutions from academia and industry towards solving the problem. Nevertheless, they still face issues including weak security guarantees, inability to scale and the coarse granularity of protection.

Two case studies demonstrated that said commercial memory safety solutions in many cases failed to guarantee the desired protection level. We saw that programmers either did not use them or were forced into a non-optimal design. From the review of state-of-the-art and the case studies, I extracted the key requirements that should be satisfied by a secure design.

As this dissertation later explores the possibility of adapting CHERI to embedded systems, I presented the background, basics and other fundamentals of capability systems and CHERI. The next chapter will explore the design of efficient compressed capability encodings and implementations for embedded space.

Chapter 3

A 64-bit compressed capability scheme for embedded systems

To adapt CHERI to the embedded space, an efficient capability processor and capability encoding must be designed and developed. Clearly, the original CHERI-256 is not a suitable candidate since it quadruples the pointer size and functions with a 64-bit address space. There are obvious opportunities for capability encoding compression, which have been attempted by previous capability machines as well as the initial versions of CHERI-128 [70]. Unfortunately, many problems still remain unresolved including software incompatibility and inability to work under typical 32-bit flat physical address space. Therefore, this chapter discusses the design and implementation of a 64-bit compressed capability format for embedded devices.

3.1 CHERI for a 32-bit machine

Current CHERI research ([75, 69, 67]) is mostly based on the MIPS-64 ISA and 64-bit FreeBSD OS. Before I began, I had conducted a quick survey to determine the typical specifications of embedded devices and to see whether the existing research artifacts directly apply. The survey was based on the WikiDevi database [72]. I looked at the *Wireless embedded system* category which was in the scope of this dissertation, with a database of 3822 devices. The survey used the custom query language provided by the database to query for all devices that were FCC approved after 2010. Devices with incomplete information (no RAM size/power specs provided) were discarded.

The survey drew the following conclusions:

1. No device had an address space that exceeded the 32-bit limit, 4GiB.

2. Many higher-end systems in this category (Raspberry Pi 3B, TP-Link TGR1900, The Amazon Fire TV CL1130) already incorporated Cortex-A class CPUs with fully-fledged operating systems. Such devices were not in scope since they were not latency and overhead sensitive. Excluding these, few had RAM sizes larger than a couple of MiBs.

3. No 64-bit processors existed before 2015 in this database. Even though 64-bit CPUs started to emerge recently (like Raspberry Pi 3B, which already deviated from the definition of embedded systems of this dissertation), almost 100% of the surveyed devices ran under a 32-bit OS, an RTOS or no operating system at all.

In fact, the ARM Cortex-R and Cortex-M series which are used in typical embedded SoCs do not contain any CPU that supports 64-bit address operations. No roadmaps and projections from ARM show any plan and necessity to introduce 64-bit processing to low-end devices. The conclusion is, embedded processors show no need for 64-bit computing at least in the foreseeable future. Therefore, instead of applying 64-bit CHERI-MIPS directly to such platforms, it is desirable to design a compressed capability format for embedded devices. Specifically, we need a low-overhead capability format for 32-bit address spaces. For smaller devices such as 16-bit or even 8-bit processors, I do not think that a capability model would be useful or practical. They are deeply embedded without communications to many other devices, thus the only attack model is still direct physical attacks, or they have extreme resource constraints that would make any extensions impractical.

3.2 Existing capability encodings

Various capability encoding schemes have been developed as capability machines and CHERI mature. In this section I review existing capability encodings and describe their merits and shortcomings to explain why a better encoding is required.

3.2.1 CHERI-256

Section 2.6.2 gives an overview of the CHERI architecture. The first implementation quadruples the pointer size by adding full 64-bit **base**, **length** fields and miscellaneous bits (Figure 2.4). The merit of this design is being straightforward in micro-architecture implementation, while its overhead is its very large pointer size. For example, Olden micro-benchmarks which exhibit an in-memory pointer density of nearly 25% (that is, 25% of the resident memory is occupied by pointers) will increase its memory footprint by almost 100%. Other common C and C++ programs with 5-10% pointer density still see their resident memory inflated by around 50%. To design the memory hierarchy, the quadrupled pointer size also suggests that the minimum bus width be 257 bits, which is not practical for small devices. Although we could get away with smaller buses and transmit a capability in multiple cycles, such a design introduces other complexities and overhead. As a result, the CHERI project is switching away from CHERI-256 in favor of new compressed capability encodings. Maintaining the 256-bit encoding is most useful for fast prototyping of new software where its simplicity in encoding outweighs the overhead.

3.2.2 M-Machine

The M-Machine is a capability model that introduces *guarded pointers*, which are tagged 64-bit pointer objects to implement capability-based addressing [13]. Guarded pointers contain 4-bit permission, 6-bit segment length and 54-bit pointer fields (**p, slen** and **pointer** respectively in Figure 3.1).

Guarded pointers are effectively a compressed capability encoding in that the **slen** field encodes the bounds (the segment) the pointer is in. **slen** is the number of bits from the Least Significant Bit (LSB) position that can vary within this segment. For example, the bottom 8 bits in the pointer field of Figure 3.1 can vary between `0x00`

Figure 3.1: *M-Machine capability encoding with example values*

and `0xff`, making the bounds of this capability `0x1200` and `0x12ff`. The hardware enforces bounded access with correct permissions.

The representability problem. Compressed capability encodings present a new difficulty that has not been seen in their uncompressed counterparts like CHERI-256. The compression schemes in this chapter all take advantage of the fact that bounds revolve around the pointer and usually share common top bits with the pointer field. These bits need not to be replicated in all the fields, therefore we can save them to achieve compression. However, this creates the problem that the moment the pointer does not share the same top bits with the bounds, it is then no longer possible to encode the original bounds. This usually happens when the pointer goes out of bounds. The problem can be demonstrated by the example in Figure 3.1. If we increment the pointer above `0x12ff`, it no longer shares the top `0x12` bits with the bounds and it is impossible to encode them.

Due to its inability to represent out-of-bounds pointers, M-Machine clears the tag when the pointer field goes out of bounds. As the capability cannot regain its tag due to monotonicity, it remains invalid forever. Even if the pointer is later brought in bounds for a valid dereference, the untagged capability remains unusable. In contrast, representability is never a problem for uncompressed CHERI-256. The base and length fields preserve full information even when the pointer goes out of bounds, which makes it still a valid capability (but invalid for dereference, of course). The capability can still be used for dereference when later the pointer is brought in bounds.

Temporary out-of-bounds pointers. Temporary out-of-bounds pointers is the behaviour in which performing integer arithmetic on pointers takes them out of their intended bounds. It is later transformed back in bounds for a valid dereference. Such behaviour pressures the representability of compressed capabilities, and breaks when a capability becomes unrepresentable.

We imagine new software stacks written with representability in mind will avoid arbitrary pointer arithmetic. However, as the CHERI compiler maintains compatibility with legacy C code, we cannot ignore that a portion of the existing codebase may exhibit such behaviour. In fact, David Chisnall et al. have investigated common C idioms which may break under a capability machine. Unfortunately, he finds a large portion of the C programs contain idioms that may lead to such breakage [15]. *ffmpeg, FreeBSD libc* and *tcpdump* are among the many common C codebases that produce in-

valid intermediate pointer results, bit-mask pointers, perform arbitrary pointer arithmetic and so forth, and each manipulation may result in a temporary out-of-bounds pointer. Without proper support for temporary out-of-bounds pointers, it is rather difficult to incorporate a large corpus of legacy C code into a compressed capability machine.

Coarse-grained objects. In addition to representability and out-of-bounds problems, the M-Machine specifically demonstrates the difficulty to encode arbitrarily-sized objects. The **slen** field is only capable of representing power-of-two regions, which means heap objects, for example, must be rounded up to fit in a capability. In the worse case, an object slightly larger than the alignment is rounded up to the next power-of-two, wasting almost 50% of the memory. In fact, this worst case might not even be rare. Heap allocators with inline metadata will add extra bytes to user requested sizes, which makes perfect power-of-two allocations always exhibit the worst case in practice. The amount of wasted memory, *memory fragmentation*, is studied later in this chapter.

Other compatibility issues. M-Machine does not support a full 64-bit address space and argues that 54 bits is sufficient. However, applications frequently rely on a full pointer field for NaN boxing [10], top-byte metadata [50] and so forth. Not preserving a full pointer breaks compatibility for these applications. Due to all the above issues, we have not seen M-Machine run any modern OS or a reasonably populated userland, and we can expect difficulty in porting existing codebases.

3.2.3 Low-fat

Low-fat is another compressed capability scheme with a similar approach to M-Machine by using top bits of the pointer to encode bounds and metadata [39].

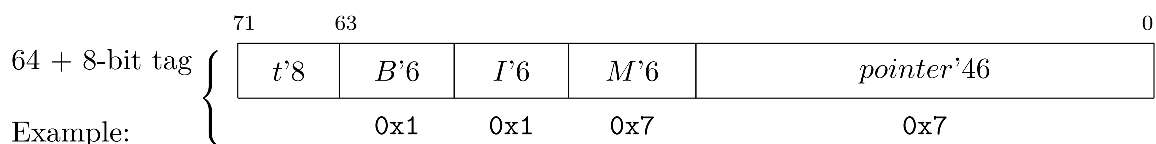
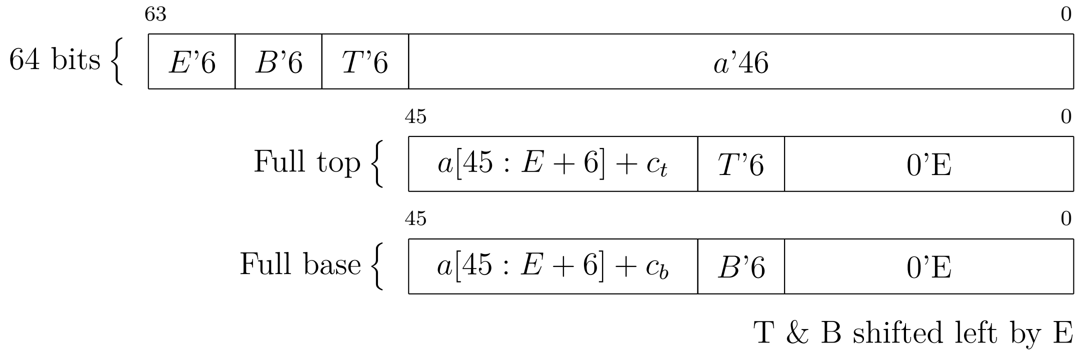


Figure 3.2: *Low-fat capability encoding with example values*

Low-fat improves granularity and precision of bounds by removing the permission field and some bits of the pointer for bounds. The **BIMA** encoding gives the block size shift **B**, the start index **I** and end index **M**, and the address field **A**. Figure 3.2 indicates a **B, I, M, A** of 1, 1, 7, 7 respectively. A block size of 2 (1<<**B** = 2) gives a region from `0x0` to `0x80`, evenly divided into 64 (6-bit **I, M** fields) 2-byte blocks. Within this region, the start and end indices further restrict the capability between `0x2` and `0xe`, spanning across **M-I**=6 blocks. Compared with M-Machine, a precision of 6 bits in the **I** and **M** fields reduces the average wasted memory to $\frac{1}{2^{(6-1)}} = 3.125\%$.

Bounds arithmetic. As the bounds arithmetic of Low-fat forms the basis of many compressed capability schemes today, it is therefore necessary to introduce the decompression algorithm to extract the original bounds. Note that the Low-fat **BIMA** terminology is rather confusing at times, so we use exponent **E**, base **B** and top **T** which conceptually correspond to block size, start and end index fields. This new terminology is used from now on throughout the rest of the thesis.



$$
\begin{aligned}
E &= \text{exponent} \\
T \ \& \ B &= \text{top \& base fields (or end and start index for Low-fat)} \\
a &= \text{address}
\end{aligned}
$$

$$
\begin{aligned}
A_{\text{mid}} &= a[E+5:E] \\
c_{\text{t}} &= (A_{\text{mid}} \geqslant T)?\ 1:0 \\
c_{\text{b}} &= (A_{\text{mid}} \leqslant B)?\ 1:0
\end{aligned}
$$

Figure 3.3: *Low-fat bounds decompression*

Figure 3.3 shows how the full base and top are decompressed. The **T** and **B** bits are left shifted by the amount of E (padded with zeros) before being masked with the pointer field. We cannot simply take the top bits from the pointer, however, because a carry bit may occur. If the **T** bits are smaller than the corresponding bits in pointer, then there must be a carry bit above the most significant bit in **T**

because by definition the full top field must be greater than the pointer. Therefore, we might need to increment the top bits, $a[45 : E + 6]$, by 1. The same happens with reconstructing the **B** field where the top bits might need to decrement by 1.

Shortcomings of Low-fat. Low-fat shares many similarities with M-Machine as well as its shortcomings. Being unable to represent out-of-bounds pointers, Low-fat also fails to be compatible with common C idioms and a wide range of existing C code. It also re-purposes permission bits for additional precision, losing the benefit of hardware permission checking. Like M-Machine, the truncated pointer breaks applications relying on full pointer fields.

In addition, the Low-fat hardware implementation is not a conventional RISC architecture and creates pipelining problems. It performs bounds checking on pointer manipulation but not on dereference. To make sure out-of-bounds does not happen, it explicitly forbids offset addressing on pointer dereference, which means the ISA does not have any efficient addressing modes. Low-fat also has a high load-to-use delay, as the register file holds partially decompressed capabilities and each capability load requires partial decoding. Worse, each capability pointer manipulation requires re-computing the distances to the base and top. The re-computed fields then need to be installed back to the partially-decoded register file.

In the end, we see the same situation as M-Machine where existing codebases are simply not portable and no major software stack has been specifically written for this particular architecture.

3.2.4 Project Aries

Another compressed capability scheme by MIT, Project Aries, extends the length to 128 bits for an untruncated 64-bit address and more flexibility for top and bottom fields, permission bits, etc [11]. This format enforces fine-grained access control with 16 permission bits and similar precision as the low-fat pointer leaving 32 unused bits for future extensions.

The encoding is shown in Figure 3.4. It inspires first versions of Low-fat, hence the similarities. Still, the problem of compatibility with out-of-bounds pointers is yet to be addressed. What again is lacking is the consideration for existing RISC

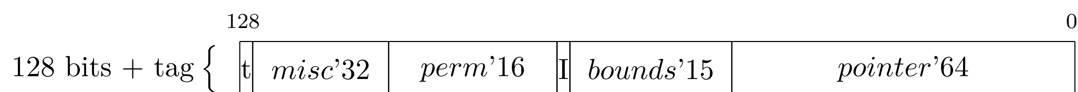


Figure 3.4: *Aries capability encoding*

architectures and C corpus. Like what we have seen before, this is a theoretical proposal and a draft implementation at best.

3.2.5 CHERI-128 Candidate 1

Before the research work in this chapter, early prototypes of CHERI-128 have been implemented by Jonathan Woodruff and Alexandre Joannou [70]. The initial implementation includes **toTop** and **toBase** to encode the distance from the pointer to the bounds. This idea encounters several problems in practice. For example, each pointer manipulation must change the distance fields and re-encode the capability in the register file. Another problem described in detail in [35] is encoding space wastage. As the size of the object gets larger, the actual top and base fields get further apart and it becomes increasingly difficult to represent out-of-bounds pointers. Nevertheless, this attempt is the first implementation of 128-bit capabilities for 64-bit address space, and raises important questions like pipelining difficulties, representable buffers and encoding efficiency which guide the design of later improved encodings.

3.2.6 Summary

I have reviewed multiple past capability schemes to this point. Unfortunately, their shortcomings pose major obstacles towards a practical capability machine. Until now, none of the aforementioned compressed capability schemes have included a rich and mature software stack that supports the architecture, and it is unsurprising that they have not seen wide deployment in production. A practical compressed capability scheme must interact well with modern RISC pipelines, incorporate existing C codebases, respect common language idioms, and show reasonable and justifiable overhead. In addition, no previous attempts focus on whether it is feasible to develop a capability scheme for a 32-bit physical address space in typical embedded processors. Such scenario, for example, requires much higher compression whose impact especially on memory is not well understood.

3.3 The CHERI Concentrate (CC) and 64-bit capability encoding

Jonathan Woodruff, Alexandre Joannou and myself aim at solving the obstacles of compressed capability schemes by developing the CHERI Concentrate format. CC learns from past mistakes, fits in a traditional RISC pipeline, improves precision of the compressed bounds and is compatible with the majority of legacy codebases. Two most important products of this research, CC-128 and CC-64, have already been implemented in hardware. In addition, they have LLVM/Clang compiler support and run 64-bit CheriBSD (FreeBSD extended with capability support) and 32-bit CheriRTOS (a capability-aware Real-Time Operating System) respectively. The two systems have demonstrated the ability to host a large number of C programs compiled under multiple capability ABIs with reasonable overhead, proving that CC is an approach towards a practical compressed capability machine. This section focuses on the design choices, novelties and key ideas of the encoding, including the study of memory fragmentation under high compression, optimisations for improved encoding efficiency and the necessary CHERI semantics to support legacy code.

3.3.1 Balancing between precision and memory fragmentation

It is safe for previous CHERI-128 work to largely ignore the memory fragmentation problem, since the number of precision bits (bits in top and base fields) is sufficiently high. For 20-bit precision in CHERI-128, objects can be precisely represented up to 1MiB. For objects that are aligned to a page boundary (e.g., mmap, file system blocks), CHERI-128 can precisely describe up to 4GiB of such large memory objects, thus the memory fragmentation issue is negligible. However, developing a 64-bit capability format for 32-bit addresses makes this assumption no longer true. Before the design of the CC-64 encoding, I would like to understand and evaluate the memory fragmentation problem to see the impact of high compression. This evaluation helps to make an informed decision in later sections.

The fragmentation problem

Compressed capabilities cannot precisely represent all bounds. Memory allocators need to ensure that compressed bounds do not improperly allow access to adjacent objects. This manifests itself as increased memory fragmentation because memory allocators may need to overallocate space and overalign objects to enforce it.

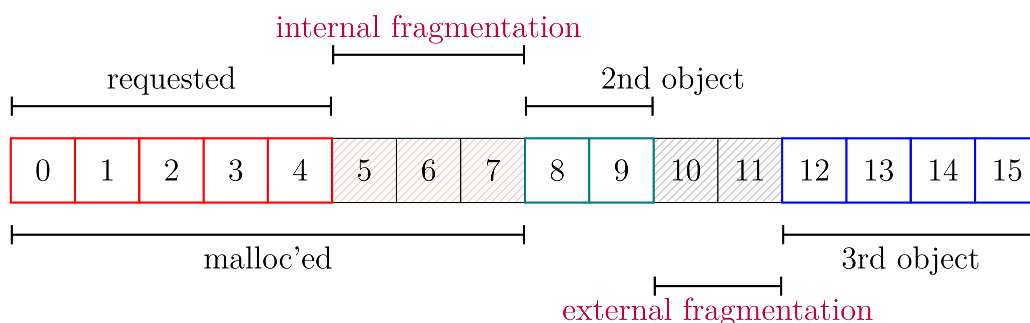


Figure 3.5: *Internal vs. external fragmentation*

Figure 3.5 illustrates memory fragmentation caused by the loss of precision, assuming a model in which capabilities are only able to represent power-of-two sizes and power-of-two aligned addresses. The figure demonstrates two typical categories:

- Internal fragmentation, caused by rounding up and padding objects so that a compressed capability can represent it.
- External fragmentation. Objects have to start from aligned addresses instead of being tightly packed.

In practice, different forms of allocation can manifest as either internal, external or both. Two most common allocation patterns in C language are of interest here, namely stack and heap allocations. The symptom of heap fragmentation is mostly internal, as the gaps can usually be filled with small allocations later. However, the C stack differs substantially due to its Last-In-First-Out (LIFO) nature. It is impossible to reuse externally fragmented space across stack frames, exposing both internal and external fragmentation problems to memory allocation. These two allocation patterns best demonstrate how much memory overhead a compressed capability encoding has under certain precisions, and will be investigated in this study.

Evaluation methodology

I evaluate fragmentation by collecting memory allocation traces and replaying them under different precisions. I thank Jonathan Woodruff for kindly providing these traces. The heap traces are from six real-world applications (Chrome 38.0.2125, Firefox 31, Apache 2.4, iTunes 12, MPlayer build #127, mySQL 5). Chrome and Firefox are traced when viewing pre-determined web pages on BBC, Facebook and Gmail. MPlayer plays Big Buck Bunny in h264@1080p. Application iTunes plays pre-determined trailers from its video store. Apache and mySQL work as the frontend and backend respectively for a web server, which is traced when running the Apache HTTP server benchmarking tool, ab. DTrace is used to hijack function calls from these applications and to filter out `malloc()` related ones. To take care of nested allocators, the initial trace is processed so that a stack of `*malloc()` calls only shows up as one entry in the final trace file. For stack allocations, we instead compile SPEC-CPU2006 benchmarks (bzip, gobmk, mcf, sjeng and a synthetic random benchmark) and hijack `CSetBounds` instructions on the stack to locate stack objects. We cannot reuse the real-world applications for stack tracing as they are too large to be ported to CHERI now, and stack tracing relies on observing `CSetBounds` instructions. For all SPEC benchmarks, we use the reference datasets as input workloads.

I extend two commonly used allocators, `dlmalloc()` [40] and `jemalloc()` [28], with additional rounding and alignment routines to ensure that every memory chunk returned by `malloc()` is precisely representable by a capability and that they are placed at sufficiently aligned addresses. The extended allocators are also parameterisable, capable of handling precision (the number of bits in the start and end index in Low-fat terminology) from 1 to 64. For the stack traces, I build a stack allocation simulator in C++ to simulate stack object placement and stack frames. Again, the simulator is aware of capability precisions and can be tuned from 1 to 64 bits. The extended heap allocators report the average internal fragmentation, as the external one is less interesting in this case, whereas the stack simulator reports the peak stack size, capturing both internal and external fragmentation.

Results

The results are presented in Figure 3.6 and 3.7. For comparison, I added the original CHERI-128 (20 bits) to the graphs.

M-Machine struggles with power-of-two bounds (equivalent to 1-bit precision) as the heap overhead approaches 30-40% in all applications. The fragmentation quickly drops with more precision, and Low-fat (6 bits) already shows tolerable overhead. All the curves are almost flat after 8 bits, and CHERI-128 has no problem precisely representing almost every object. The results also confirm that heap allocators already round up and align objects to ease management, as even the perfect precision shows some internal fragmentation. `jemalloc()` apparently has stronger alignment requirements due to its need to accommodate paged-memory in modern machines.

The nature of the stack exacerbates the problem for low precisions. Both the results and visualisations of the stack allocations suggest that external fragmentation is contributing to high peak stack sizes. The total size of stack allocations is usually small for each benchmark (under 64KiB), therefore even a 200% overhead would not be a problem compared with heap allocations. However, it indicates that for low precisions, other allocations that share the same pattern (unable or difficult to reuse external fragmentation, e.g., large number of separate small sandboxes within a process) could waste a large amount of memory space when facing a similar problem.

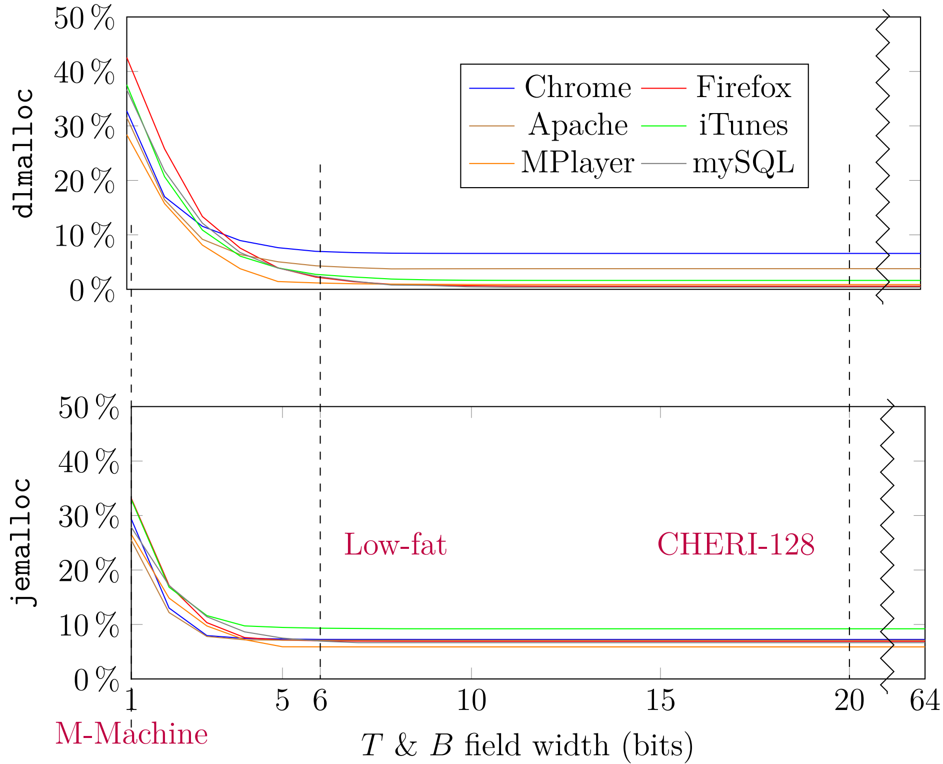


Figure 3.6: *Heap internal fragmentation*

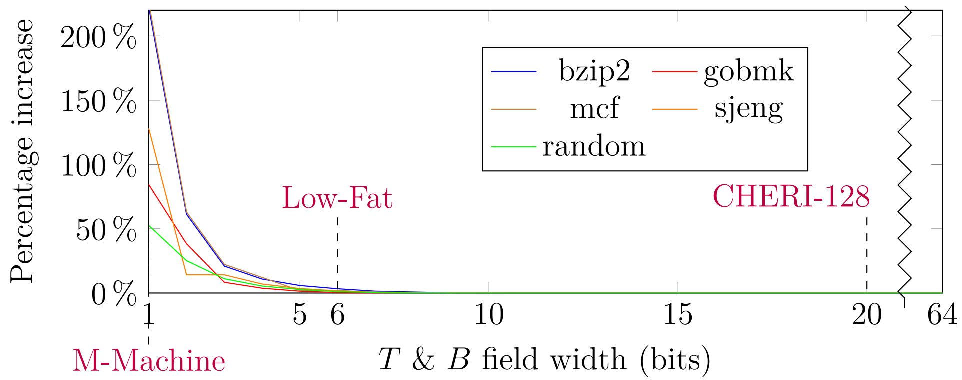


Figure 3.7: *Percentage increase in peak size of total stack allocations (SPEC CPU 2006 experimental builds)*

Analytical model

Stack allocations from the benchmarks are mostly small objects with a few large objects dominating the overhead, thus the peak stack size is highly dependent on how the workloads allocate large objects on the stack. However, the number of heap allocations is usually significantly higher with a mixture of objects of various sizes, whose average can be summarised with an analytical model.

For fragmentation in Figure 3.6, its upper bound can be calculated with respect to the precision. A precision of n divides the maximal possible object under a certain exponent (2^{n+E} bytes) into 2^n blocks. In the worst case, we need to pad the object with almost a whole block (2^E bytes) to account for the lack of precision. Also, the object can be as small as slightly over half of the largest possible object under its exponent, which is 2^{n+E-1}. Therefore, the worst case internal fragmentation is:

$$Internal\ fragmentation = \frac{2^E}{2^E + 2^{n+E-1}} = \frac{1}{1 + 2^{n-1}}$$

If we assume the sizes of allocations are approximately uniformly distributed overall in evaluated applications, the average object size is the average of the largest and the smallest under a certain exponent:

$$Avg.\ obj.\ size = \frac{2^{n+E-1} + 2^{n+E}}{2} = 2^{n+E-2} + 2^{n+E-1}$$

and assuming the required padding on average is halved (2^{E-1}) as well, the average internal fragmentation will be:

$$Avg. = \frac{avg.\ padding}{avg.\ padding + avg.\ obj.\ size} = \frac{2^{E-1}}{2^{E-1} + 2^{n+E-2} + 2^{n+E-1}} = \frac{1}{1 + 2^{n-1} + 2^n}$$

Of course, additional heap metadata must be allocated as well for heap maintenance in practice, which often disrupts how objects can be placed or aligned, resulting in higher-than-predicted fragmentation. This is much more visible in `dlmalloc`, since its boundary-tagging design must embed inline metadata in every allocation [40],

potentially inflating objects to the next alignment boundary or even to the next exponent.

At high precisions, the padding of `dlmalloc` is small compared with the inline metadata, thus the internal fragmentation mostly reflects the wasted space from metadata, which we cannot eliminate no matter how high the precision is. Since the inline metadata of each allocation is constant regardless of the object size, the ratio of internal fragmentation at high precisions depends on the allocation pattern. An application performing a large number of small allocations requires more memory for metadata than one with a small number of large allocations, explaining why some applications can reduce fragmentation to almost 0%, thus fitting the model nicely, while others cannot.

Unlike `dlmalloc`, `jemalloc` is designed for paged systems and performs its own rounding internally, effectively only having a precision between 3 and 6 for medium objects and 3 for large objects [28]. As the curve shows, it quickly becomes flat after $n = 3$, and approximately matches the equation between $n = 3$ and $n = 6$ for all high precisions.

Summary

The internal and external fragmentation can be captured nicely in the C stack and heap, representing the most common sources of memory overhead for C codebases. Parameterisable tools help visualise the tradeoffs between consumed precision bits and memory wastage. The evaluation also brings the discussed compressed capability schemes together for comparison. M-Machine represents one extreme where fragmentation is unacceptably high, whereas CHERI-128 shows almost no concern about memory overhead issues. Low-fat (6-bit start and end index, 6-bit block size, 18 bits total for bounds) might be a good balance for 64-bit capabilities. For some extra headroom, my initial plan is to dedicate 22 bits in total for compressed bounds with 8-bit precision.

3.3.2 Improved encoding efficiency

A capability with certain bounds under Low-fat can be encoded in different ways. The example in Figure 3.2 encodes the block size, base, top and pointer of `1`, `0x2`,

`0xe` and `0x7` respectively. Clearly, it is possible to use a block size of 1 and assign `0x2` and `0xe` to **I** and **M** to encode exactly the same bounds. The same capability being represented in multiple formats is wasting encoding space. To save encoding space, we pick the canonical to be the minimum block size possible to encode the bounds. In the Low-fat example, a block size of 1 is the minimum required and should be picked instead of 2. Choosing the minimum makes sense because it also gives the most fine-grained bounds. If the object size in Figure 3.2 is 11 instead of 12, then a block size (or exponent) of 2 cannot precisely represent the bounds and must be padded to 12, leading to memory fragmentation problems. In the CHERI ISA, this is enforced by always choosing the minimum block size (or exponent, **E**) in `CSetBounds`, which is the only instruction that is able to modify it.

Implied most significant bit in T

Take a 6-bit **T** and a 6-bit **B** as an example, we imagine another 6-bit field, **L**, which is derived from **T=B+L**. By always choosing the minimum **E**, it is guaranteed that the top bit of **L** is 1, because otherwise, the object size is smaller than half of the largest object under this exponent, therefore a even smaller **E** can be chosen. This observation is important, as it allows us to derive the top bit of **T** from the remaining bits in **T** and **B**.

```
LCarry = 0
if T[4:0] < B[4:0]:
  LCarry = 1
T[5] = B[5] + LCarry + L[5] (L[5] must be 1)
```

As $T[5]$ can be derived, we are able to save a bit in **T** for other purposes. Note that the $L[5] = 1$ assumption can only be made when **E** is non-zero. When **E** is zero, the object size can be arbitrarily small and $T[5]$ has to be preserved.

Internal exponent

Another novelty incorporated into the final CC format is the internal exponent. For all the compressed capability encodings reviewed so far, they devote equal number of bits to **T**, **B** and **E** for objects of all sizes. However, the precision requirements of different sizes are inherently different. More specifically, objects tend to have alignment requirements in practice, and larger objects tend to be better aligned, requiring

less precision. Examples include padding structs and functions for performance, directly calling `mmap()` on the heap for large allocations, 1 MiB alignment for disk partitions, well-aligned memory windows for memory-mapped devices, etc. In fact, the paper on jemalloc [28] makes a similar observation that most allocations happen to objects under 512 bytes. Therefore, if we are flexible about the number of bits devoted to different precisions, optimisations can be made to reduce overall memory overhead.

Initially, we developed a run-length encoding for the exponent so that a smaller **E** (i.e., small objects) consumes fewer bits than a larger one, leaving more space for **T** and **B** bits. This naturally fits the previous observation, but comes at a huge cost in decoding complexity. Eventually, we use the saved bit from the implied most significant bit in **T** to construct a much simpler encoding, presented in Figure 3.8. To be comparable with Low-fat for evaluation, we keep 6 bits of precision here.

8		0
I_E	$T[7:3]$	$T[2:0]$ or $E[2:0]$
$B[8:3]$		$B[2:0]$ or $E[5:3]$

Figure 3.8: *Improved Low-fat Encoding with Embedded Exponent and Implied* T_8

The encoding demonstrates the internal exponent idea. The saved implied bit I_E now indicates whether **E** is zero. If so, there is no need to have a separate **E** field, and we have a full 9-bit **B** and 8-bit **T**. Remember that the top bit of **T** can no longer be implied when **E** is zero, thus the effective precision is 8 bits not 9. When I_E is set (i.e., the exponent is non-zero), the **E** "steals" 3 bits from **T** and **B** respectively to form an actual 6-bit exponent, hence the name *internal exponent*. Compared with Low-fat, CC is a strict improvement and never performs worse. It specifically improves the precision for small objects and can represent objects precisely up to $2^8 = 256$ bytes instead of $2^6 = 64$ bytes for Low-fat.

Evaluation of representability

With the improved encoding, I rerun the benchmarks to report the percentage of imprecise capabilities.

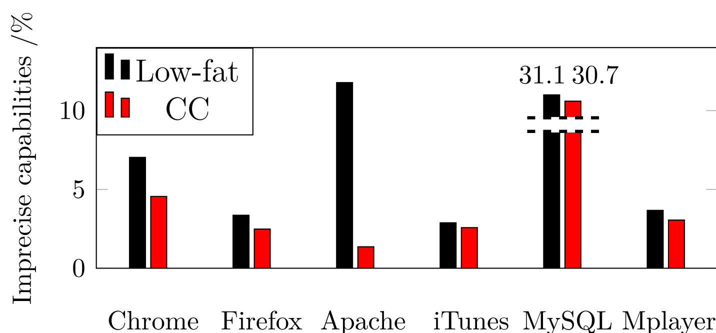


Figure 3.9: *The percentage of allocations that cannot be precisely represented in a capability. Lower is better.*

Figure 3.9 shows that CC never performs worse than Low-fat. The effectiveness of the improved encoding depends highly on the nature of allocations in an application. Most benchmarks benefit moderately while Apache reduces imprecision drastically, revealing the fact that it does a large number of allocations for small objects. So far, we improve the encoding efficiency without adding bits for bounds or introducing any new semantics compared with Low-fat, and only add minor decoding complexity to bounds.

3.3.3 The CHERI semantics

The major roadblock towards wider adoption of compressed capability machines is the issue with legacy codebases. This section introduces necessary new semantics to ensure compatibility.

Full pointer fields

Truncating the pointer field to 46 bits to make room for bounds in Low-fat creates a non-traditional address space. Such a decision violates many design patterns and C language idioms in traditional software. For example, top-byte metadata [50] allows metadata to be stored in the unused upper bits (which do not participate in memory access and address translation) of a pointer, which breaks with a truncated pointer. Similarly, the AArch64 architecture provides the top-byte-ignore feature that permits the OS and applications to compress information into unused pointer bits. For common C idioms, the casts between pointers and integers often require the pointer

to preserve full information of an integer, and conventional architectures without the distinction between pointers and integers exacerbate the problem that programmers tend to use integer and pointer types in C interchangeably.

For 32-bit embedded systems, even if pointer metadata is uncommon and RAM is small, it does not necessarily mean the address space is much smaller than the full 4GiB. Memory mapped devices and registers are commonly scattered in the physical address space. Even if the entire address space is small enough to make extra bits available, using them to encode capabilities also prohibits any future expansion of the address space like adding new memory-mapped devices.

Clearly, if all such design patterns, idioms and address space configurations can be avoided, it is advantageous to repurpose the unused pointer bits for capability encoding, especially increasing the precision of compressed capabilities. However, in this research we find that fixing all the issues to work with a truncated pointer field is infeasible in practice. As a result, we decide to keep a full pointer field in favor of much better compatibility, which means a 32-bit address space for 64-bit capabilities. Reducing the pointer to 32 bits also saves 1 bit from the **E** field, which is later used to provide out-of-bounds buffers to allow temporary out-of-bounds pointers. For 64-bit address spaces, CHERI-128 is used to preserve full pointers.

Permission bits

The low precision in M-Machine is not acceptable, while removing permission bits to encode more precise bounds in Low-fat creates other confusions, such as its inability to distinguish between read and write or code and data capabilities. The latter, for example, is a serious issue as it is impossible to enforce $W \oplus X$ on the capability level. We acknowledge that it is necessary to include permissions bits to guarantee any meaningful memory protection.

Returning to a traditional address space leaves half of the bits in a capability, enough to encode sufficient permission and bound bits. CHERI compressed format conflates all the permission bits for privileged system registers into a single **access_sys_regs** bit, and other permissions are the same as CHERI-256. In the end, a full pointer with permission bits means CC capabilities double the size of a traditional integer pointer. However, we believe the importance of compatibility and fine-grained permission control far outweighs the overhead it incurs.

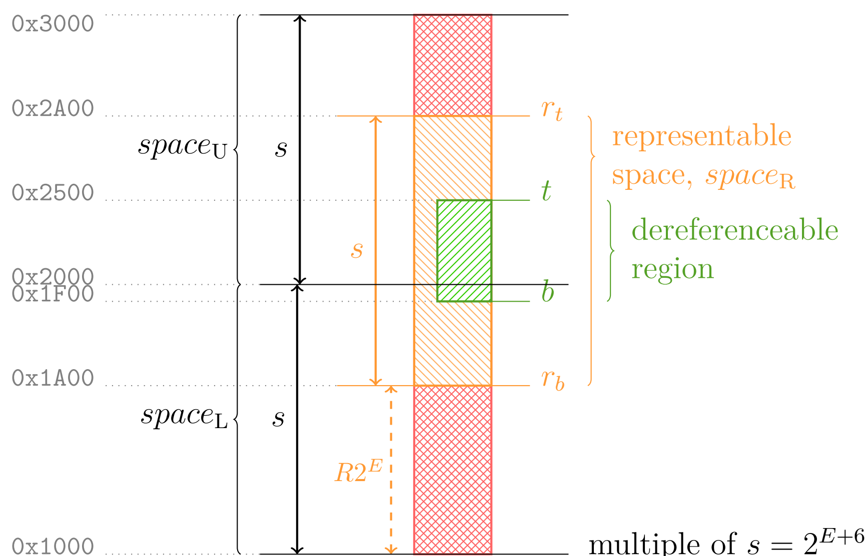


Figure 3.10: *CHERI Concentrate bounds in an address space. Addresses increase upwards. The example shows a `0x600`-byte object based at `0x1F00`.*

The representable buffer

In compressed capability schemes, it is impossible to arbitrarily change the value of the pointer, as it may also change the bounds that are partially reconstructed from the pointer field (like in Figure 3.3), which leads to the inability to support arbitrary out-of-bounds pointers. The M-Machine and Low-fat approach to simply invalidate the capability the moment it goes out of bounds is unacceptable. To overcome the problem, we add a *representable buffer* space, which the pointer field can point to without changing the decompressed bounds, even though the pointer may have already gone out of bounds.

An example is presented in Figure 3.10, assuming 6-bit precision and an exponent of 7. For simplicity, the example does not incorporate implied MSB in **T** and the internal exponent. The example categorises the space into three regions. The dereferenceable region indicates the actual bounds of the capability. The representable region is where the pointer field can reside without affecting the representability of the original bounds. In the example, the pointer can change between `0x1A00` and `0x2A00`. The representable region (orange) cannot be larger than s, because if so, there will be two pointer values with the same $a[E+5 : E]$ but different $a[: E+6]$ (see Figure 3.3), which will then be decoded into two different pairs of bounds. Obviously, the representable region has to be a superset of the dereferenceable region. Outside

is the unrepresentable region. Whenever a pointer field is modified into this region, it is no longer possible to decode **T** and **B** into the original bounds, and the capability has to be invalidated.

Figure 3.10 denotes the lower and upper bound of the representable region with r_b and r_t ($r_t = r_b + s$). Let us define **R** to be $r_b[E+5:E]$ ($r_t[E+5:E]$ is the same), and **A** to be $a[E+5:E]$. To reconstruct the full bounds, carry bits must be taken care of. Low-fat is simpler because the pointer must be within bounds. Adding the representable buffer means the pointer can be greater than the top or lower than the base, complicating the decoding of carry bits.

$$c_t = sgn(sgn(R - T) - sgn(R - A))$$

$$c_b = sgn(sgn(R - B) - sgn(R - A))$$

$$sgn(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x = 0) \\ -1 & (x < 0) \end{cases}$$

The comparisons among **R**, **B** and **T** calculate whether they are in the same s space. If not, a carry of $+1$ or -1 must occur. With the new carry bit calculations and the same shifting and masking logic in Figure 3.3, the bounds can be uniquely decoded as long as the pointer is in representable region.

Of course, a representable buffer does not entirely solve the out-of-bounds pointer problem. The capability remains valid as long as the pointer field is in the representable region (it does not have to be within bounds). However, if the pointer field goes too far and enters the unrepresentable region, CC still faces the same problem as the capability is invalidated forever. Nevertheless, a properly-placed and sufficiently-large buffer region can eliminate almost all temporary out-of-bounds issues. In the porting of CheriBSD, only few corner cases needed to be fixed after introducing the representable buffer.

3.3.4 CC-64

Incorporating all the aforementioned techniques, the final CC-64 format is implemented as in Figure 3.11.

31						0
p'10	I_E'1	$L[9]$'1	$T[8:2]$'7	T_E'2	$B[10:2]$'9	B_E'2
a'32						

Figure 3.11: *64-bit CHERI Concentrate*

The CC encoding in Figure 3.11 highlights the improvements and novelties discussed before. This design chooses to add a representable buffer that is at least as large as the object itself while still selecting the minimum **E** possible. It means the size of the object is always $\frac{s}{4} \leqslant size < \frac{s}{2}$ ($s = 2^{E+11}$ here) for non-zero **E**. The rest of the space in s, $\frac{s}{2} < rest \leqslant \frac{3}{4}s$, is the representable buffer space. Hence, the **L** in **T=B+L** must be `b'01` in the top two bits, leaving the top two bits in **T** to be implied. In other words, one precision bit is devoted to the representable buffer. The one extra implied bit $L[9]$ in **T** now forms part of the **E**, therefore $E = \{L[9], T_E, B_E\}$ when **E** is internal.

As RAM is usually a scarce resource for embedded systems, the number of precision bits must be sufficient to minimise memory fragmentation. My study in Section 3.3.1 suggests that 6 bits reduces the overhead to a reasonably low level and 8 bits is safer. CC-64 has 8 bits for non-zero **E** (11 bits in **B** in total, minus 2 bits for the internal exponent and 1 bit for the representable buffer) and 10 bits for a zero **E** (11 minus 1 bit for the buffer), precisely representing objects of up to $2^{10} = 1KiB$ and providing enough precision beyond that. Further, embedded workloads which adapt to memory-constrained scenarios tend to do smaller allocations and allocate less frequently, which is exactly what the CC encoding is optimised for.

Detailed arithmetic for encoding and decoding CC-64 can be found in Appendix A.

3.4 Summary of compressed capability schemes

Table 3.1 summarises the capability encodings.

Despite the advantages of CC-64 described in the table, the doubled pointer size is undesirable and may result in a higher memory footprint and cache pressure. Nevertheless, this overhead directly correlates to the pointer density in memory. The paper [74] finds at low density levels, the overhead is negligible even with the purecap

Table 3.1: *Comparison of capability encodings*

Encoding	Size	Addr.	Precision	Max. frag	× ptr.	Perm.	Full ptr.	OoB
CHERI-256	256	64	64	0	4×	✓	✓	✓
M-Machine	64	54	1	300%	1×	✓	×	×
Low-fat	64	40	6	3.125%	1×	×	×	×
CHERI-128 C1	128	48	16	<0.01%	2×	✓	×	buffer*
CC-128	128	64	20 or 22	<0.01%	2×	✓	✓	buffer
CC-64	64	32	8 or 10	0.78%	2×	✓	✓	buffer

Addr. = number of address bits, Max. frag = maximum possible memory fragmentation, × ptr. = size compared with an integer pointer, Perm. = permission bits, OoB = temporary out-of-bounds capabilities, *the representable buffer size depends on the object size

ABI (all pointers are capabilities). For embedded systems where the pointer density is low and pointers are only selectively replaced with capabilities, the overhead is expected to be negligible.

3.5 A CC-64 hardware implementation

This section explores whether the CC-64 algorithms lead to a feasible implementation. To understand the overhead, I compare it with the dominant embedded system memory protection device, the MPU.

3.5.1 The MPU model

Section 2.3.2 has already provided a high-level overview of how MPUs protect physical memory. I choose to look at the RISC-V PMP, as other MPU models from ARM or Intel do not publish an open-source specification or implementation details. I follow the RISC-V specification in [66] for architectural and micro-architectural implementation. Generally, the PMP enables multiple protected regions, which are defined by bounds and protection attribute registers. The following demonstrates the pseudocode to determine whether a memory request can be granted:

```
// n = number of regions
// addr = address of mem request
// size = size of mem request
```

```
// region = the array of protected regions
// acc = read, write or execute access
bool check_access(addr, size, acc):
  permitted = false
  for i in 1 to =n:
    matched = match(addr, size, region[i].bounds)
    if matched:
      permitted = check_perms(acc, region[i].perms)
  return permitted

// auxiliary functions
bool match(addr, size, bounds):
  if addr>=bounds.lower and addr+size<bounds.upper:
    return true
  return false

bool check_perms(acc, perms):
  if acc in perms:
    return true
  return false
```

The algorithm is simple. The memory access is checked against all regions. If there is a match (in its bounds), the access is checked against the permissions of the region, which eventually grants the access. Note that this algorithm essentially grants priorities to higher-numbered regions. That is, if a match occurs to two regions, the permissions of the region with a higher index override those of a lower region.

A hardware implementation is straightforward following the pseudocode. An obvious optimisation is that the for loop in `check_access()` can be implemented in parallel, matching all regions at once. However, I still have to serialise the match results due to the effect of higher regions overriding lower ones.

RISC-V dictates that two modes be supported in the `match()` function. When power-of-two is enabled, each region is defined by a bounds register and a permission register. The bounds register embeds a bit mask in a pointer, and a match occurs if the top bits of the memory access address match the masked pointer [66]. This is essentially the same mechanism with the M-Machine where a shift value indicates

power-of-two bounds. The base-bound mode combines two bounds registers into one region, specifying the base and the length in two registers, which trades off half of the number of regions for much finer granularity.

I implemented the PMP in Bluespec SystemVerilog® and incorporate it with the baseline BERI processor to offer physical memory protection. To match the most common configurations, 8 base-bound regions (16 power-of-two) are included.

3.5.2 Capability coprocessor vs. MPU

To understand how CC-64 competes with the MPU, I implement CC-64 in the 64-bit capability coprocessor. I synthesise two CPUs, one enhancing BERI with the CC-64 capability coprocessor and the other adding the PMP unit. The original BERI CPU is heavily refactored to reduce the address width from 64 to 32 bits. To match the PMP, the capability coprocessor also has 8 general purpose capability registers. To make the costs clear, I disable BRAM usage so that all logic is generated using combinational circuits or registers on the Stratix® IV GX FPGA. I synthesise 5 times and take the mean and deviation. The Bluespec code of the CHERI-64 coprocessor has been merged into our CHERI repository.

Table 3.2: *FPGA resource utilization and timing*

Category	PMP Unit	Capability Coprocessor
Total ALUTs	$5174 \pm 0.5\%$	$7212 \pm 0.4\%$
Combinational w/o register	$4241 \pm 0.6\%$	$3879 \pm 0.6\%$
Combinational w/ register	$759 \pm 2.6\%$	$2278 \pm 0.8\%$
Register only	$174 \pm 11.5\%$	$1055 \pm 1.9\%$
Core clock freq. (MHz)	$86.71 \pm 7.3\%$	$105.83 \pm 3.0\%$

Area. The logic utilisation is shown in Table 3.2. Overall, the capability coprocessor has 39.4% more logic utilisation. Note that the capability coprocessor not only supports capability registers, but also implements the CHERI ISA; the PMP is a basic implementation of the RISC-V specification. I anticipate that any extensions on PMPs or MPUs will quickly increase the logic utilisation. These include increasing the register count to 16 to enable more regions, the Execution-Aware MPU from TrustLite and TyTAN, separating MPU entries further into sub-regions as in ARM

embedded processors, etc. Therefore, a well optimised capability coprocessor should not be much more expensive than a commercial MPU or PMP in terms of logic usage in an ASIC.

Critical path and timing constraints. The pipeline of the capability coprocessor is simple and operates in parallel with the main pipeline, not disturbing the critical path. However, fitting the PMP into the pipeline proves to be very difficult. For 8 PMP entries, I have to perform a full associative match, introducing 32×32-bit comparators (each PMP entry has a base and length and the associative match has to be done for both data and instruction fetch) within a single clock cycle. This difficulty is confirmed by timing analysis, which shows that the maximum clock frequency achieved is around 20MHz lower than with the capability coprocessor (Table 3.2). Unsurprisingly, further analysis reveals that the PMP lies in the critical path while the capability coprocessor does not.

Power. Although FPGA synthesis is not indicative of ASIC power, qualitative estimates can still be made. The major power draw from PMPs (as well as MPUs) is the large number of comparisons within each cycle. A capability coprocessor has drastically reduced power consumption due to the absence of associative searches. CHERI always specifies the region of each access explicitly, so that only one bounds check is required for instruction fetch, and one for data access: instruction fetch is checked against PCC[1], and the data access is checked against either the DDC or an explicit capability. On the other hand, the CHERI coprocessor does require additional power to decompress the compressed capability bounds, but this is still dramatically less than the power required by an active MPU. Note that the MPU model also has room for optimisation. For example, the EA-MPU avoids full associative searches by linking code and data regions. Once the code entry is matched, only linked data regions will be searched.

[1] A well-optimised implementation might not even need a full PCC bounds check on instructions. For example, we can just check the bounds on jumps or branches and use a simple counter to count down how far the PC is to the end of PCC when the PC advances normally.

3.5.3 Other considerations

64-bit buses

32-bit CPUs with 64-bit capabilities might require the buses to be at least 64 bits. Although tiny systems cannot tolerate the cost of this upgrade, many CPUs already have 64-bit buses like Cortex-M7 and some Cortex-R cores. This range of CPUs are often connected to other systems and therefore already incorporate memory protection, and is what this research is targeting. For this market, I do not think supporting a wider type in the pipeline or the bus is a major problem.

Tags

Existing research explores efficient tag cache designs to minimise extra DRAM traffic [34]. For small devices, the cost of a tag cache might not be acceptable. However, considering that these devices in which even a tag cache is intolerable often have only KiBs of RAM or adopt custom designs, custom solutions may be used to cater to their needs. For very limited memory, caching is simply not useful and all the tags could be stored as a bit vector close to the CPU. On the other hand, embedded SoCs often use small and custom SRAMs for memory. A custom SRAM could just extend the interface with a tag bit and add native tags to memory words, avoiding the complexities and workarounds of using off-the-shelf DRAMs.

3.6 Summary

This chapter attempted to develop a suitable compressed capability format for embedded devices. The new CC-64 format learnt from past mistakes, optimised for embedded use cases and maintained compatibility with software stacks. Also, many design choices in this chapter were informed and guided decisions, aiming at supporting a wide range of software. The FPGA implementation showed that incorporating the CC-64 capability coprocessor was not more difficult or expensive than the mainstream MPU solutions, and was definitely feasible.

The contributions so far are the groundwork. What remains to be seen is whether it supports a practical capability-aware Real-Time Operating System and applications

running under such an environment. This chapter developed an optimised hardware implementation for compressed capabilities in embedded devices, and the next chapter will explore a capability-aware Real-Time Operating System.

Chapter 4

CheriRTOS

On top of the CC-64 hardware, I implement a CHERI-aware Real-Time Operating System, CheriRTOS. The first point of this chapter is to prove that the CC-64 hardware is capable of hosting a practical system. Further, and most importantly, this work shows that it is possible to efficiently isolate tasks and domains and to protect memory at arbitrary granularities in a single physical address space. These new possibilities not only come at a low cost, but bring higher scalability, simplicity and future-proof designs as well.

4.1 A CHERI-based RTOS

Current research of incorporating CHERI into modern operating systems concentrates on the CheriBSD project [71]. CheriBSD extends the FreeBSD OS with CHERI primitives and enhances paged memory with address-space memory protection and compartmentalisation. So far, the research effort mostly resides within user space and within the same address space, as the separation between processes is guaranteed by memory translation itself. Capabilities are created, copied and dereferenced in a process but not propagated across address space boundaries. As long as the OS does not falsely allow arbitrary flow of capabilities between address spaces, a capability is an unforgeable token. As a result, the first implementation of CheriBSD requires minimal changes in the kernel, other than preserving capability registers on context switches and converting between capability and integer arguments on system calls. Ongoing research is exploring more capability-aware kernel designs. For example, the *purecap*

kernel replaces pointers with capabilities for kernel space protection; the *co-process* model confines processes to a single address space, using capability compartmentalisation rather than virtual memory for isolation; the CheriABI work extends prior user-space-only pointer protection to the OS kernel through construction and maintenance of abstract capabilities, allowing kernel control flows, data structures and compartments to be protected and isolated by CHERI [22].

For embedded CPUs with no MMU, there is no virtual memory to fall back to. The lack of MMU places all tasks together with the kernel into a single physical address space, hence the difficulty in isolation. Theoretically, embedded processors with MPUs can swap MPU registers on context switch boundaries to guarantee task isolation. However, Section 2.3.2 explains that the high latency of such an approach often prevents any fine-grained use. Therefore, I would like to investigate how a capability RTOS and applications might use capability protection for efficient, scalable task isolation as well as fine-grained memory protection.

4.2 Real-time operating systems

Embedded processors commonly have low-latency and real-time requirements from real-time applications. For tiny devices and the simplest configurations, applications are run on bare-metal for only a few dedicated and static jobs. This model is increasingly obsolete as devices are becoming more capable and the use cases are becoming more diverse, like the WiFi chips described in Chapter 2 handling WiFi traffic, real-time events and communications with the main CPU on a single Cortex-R processor. The increase in processing power, while still highly constrained, enables programmers to adopt new abstractions like real-time operating systems (analogous to other modern OSes) and tasks (analogous to processes). Although one may choose to understand RTOSes through the normal OS concepts, several key aspects are fundamentally different in an RTOS environment. For example:

Tasks

Without memory translation, the traditional abstraction of processes is impossible. Instead, tasks are the abstraction under a lightweight and pure physical RTOS. The RTOS loads the code and data from ROM, Flash or other external storage, cre-

ates the corresponding kernel data structures, adds the task to the scheduler, and when the task is ready, jumps to its entry point. Note that a task could be viewed as simply a piece of code and data in physical memory, as there are no inherent boundaries between tasks. Any task is able to view the entire memory space. Because of the security implications, RTOSes rely heavily on various security components like the MPU and PMP as described in Chapter 2 for any restriction on physical memory access.

Real-time guarantees

The priority of real-time systems is to guarantee that tasks meet their deadlines, not to optimise for raw performance. Meeting the deadlines also ensures the predictability of the system. The requirements of deadlines may vary: missing a *hard* deadline results in catastrophic consequences; missing a *firm* deadline renders the computation results unusable; a *soft* deadline is a desirable target, introducing a penalty if it is missed, but still allowing the system to proceed normally [26]. Many schedulers exist, including priority-driven, earliest deadline first, least slack time first, preemptive and cooperative scheduling algorithms [26]. These target different scenarios and tradeoff between scheduling overhead and real-timeness, and have different implementation complexities.

Low-power, interrupt-driven and deterministic

Running on batteries or low-voltage power sources, the processors of these systems usually adopt an interrupt-driven model where tasks are created or started via external interrupt signals. Combined with real-time requirements, applications developed around such a model rely on deterministic delays and timings. For instance, a task must start at a certain delay after the interrupt is fired, or the computation results must be available at a certain time after an event. Therefore, designs that work against this principle should be avoided: scheduling algorithms that have $O(n)$ complexity with respect to the number of tasks should be modified to be $O(1)$; caches must be introduced with extra care to make sure they do not add randomness to deterministic routines.

The key differences above illustrate how RTOSes could diverge from typical UNIX operating systems. The unique design requirements motivated the emergence of em-

bedded security components like MPUs and Trustzone. Such requirements are also the guidelines for the design exploration and implementation in this chapter.

4.3 Prerequisites

This section highlights the prerequisites, especially the engineering efforts to prepare for a CHERI-aware RTOS.

4.3.1 The MIPS-n32 ABI

The base MIPS-n64 ABI for BERI/CHERI does not function well with the 32-bit address field in CC-64. Instead, I adopt the MIPS-n32 as the base ABI for a 32-bit address space. The original o32 from the MIPS32 era is also an option, but it is a rather old ABI that is not actively maintained in Clang/LLVM. The handbook [29] provides a detailed tutorial of MIPS-n32.

MIPS-n32 uses 32 general purpose registers in a similar convention as n64, which is a larger register file than a common embedded configuration. In contrast, ARMv7-R/M and RISC-V rv32ec all have 16 registers. A smaller register file reduces logic utilisation and power, but more importantly, reduces the latency of context switches and interrupts. I modify n32 to approximate a typical embedded ABI as found in ARM and RISC-V processors.

Registers	Convention
$0	zero register
$at	scratch register
$v0-$v1	temporary and return registers
$a0-$a3	argument registers
$t0	temporary, caller-save
$s0-$s3	callee-save
$gp, $sp, $fp, $ra	global, stack, frame pointers and return register

4.3.2 Enabling CHERI-64 in Clang/LLVM

This part is mostly engineering effort to add CHERI-64 support in our Clang/LLVM toolchain. CHERI-64 depends on the MIPS-n32 backend in LLVM which was unfortunately buggy and incomplete. Robert Kovacsics and I investigated the issues and submitted multiple patches and many of them were merged into LLVM. The MIPS-n32 now tracks the completeness and quality of the n64 backend.

I proceed to add CHERI support to MIPS-n32, mostly porting the existing frontend and backend CHERI code to the n32 ABI. As n32 supports 64-bit integers, the compiler has to be modified to further distinguish between 64-bit integers and 64-bit capabilities. To match the hardware, the backend only has 8 general purpose capability registers to allocate, and the current convention is:

Capability registers	Convention
$c1-$c2	caller-save & CCall registers
$c3-$c5	argument registers
$c6-$c7	callee-save
$c8	sratch register, caller-save
$PCC	Program Counter Capability
$DDC	Default Data Capability
$KR1C	trusted stack register

The backend register allocation sees the 8 general purpose capability registers. There are 3 system capability registers (**PCC**, **DDC** and **KR1C**) which are for the kernel and will be elaborated later.

The compiler can be found on https://github.com/Jerryxia32/llvm/tree/n32 and https://github.com/Jerryxia32/clang/tree/n32.

4.3.3 A baseline RTOS

Before I implement a CHERI-aware RTOS, a suitable baseline MIPS RTOS must be found. Robert N. M. Watson and Hadrien Barral experimented with a pure capability microkernel, CheriOS, which exercised the idea of object capabilities and object

activations. I extract several components from CheriOS to construct a non-CHERI bare-metal environment on MIPS, including the task loader, round-robin scheduler, SoC drivers and message queues. The first version supports basic multi-tasking and inter-task communication.

The initial kernel is hardly an RTOS due to its minimum feature set. To see what qualifies as a production-level RTOS, I follow the specifications of FreeRTOS [4]. FreeRTOS is developed by Real Time Engineers, Ltd., ideally suited for deeply embedded applications on microcontrollers and microprocessors. Embedded Linux and FreeRTOS occupy two largest market shares of all embedded environments [3], and FreeRTOS is a recognised reference implementation. I implement most of the APIs in its specification, especially those crucial to real-time operations, including:

- Task priorities.
- An $O(1)$ task scheduler, configurable to support all FreeRTOS scheduling modes like time slicing, preemptive, priority-driven, cooperative, etc.
- Dynamic centralised heap manager.
- Cycle-accurate timer APIs.
- Direct user space task notification to avoid the delay of message queues.
- Dynamic task loading.

Improving the completeness of real-time APIs not only qualifies this baseline system as an RTOS, but eases the porting effort of applications to this system as well. The baseline pure MIPS RTOS compiles with the patched MIPS-n32 compiler and runs on the 32-bit BERI described in Section 3.5.2.

4.4 CheriRTOS

After implementing RTOS APIs, I built CheriRTOS on top of the MIPS system. Instead of deploying MPU, SAU or Trustzone for security, the kernel is aware of CHERI and has access to capability registers. It offers fine-grained memory protection, task isolation, fast domain transition, return guarantee and real-time guarantee,

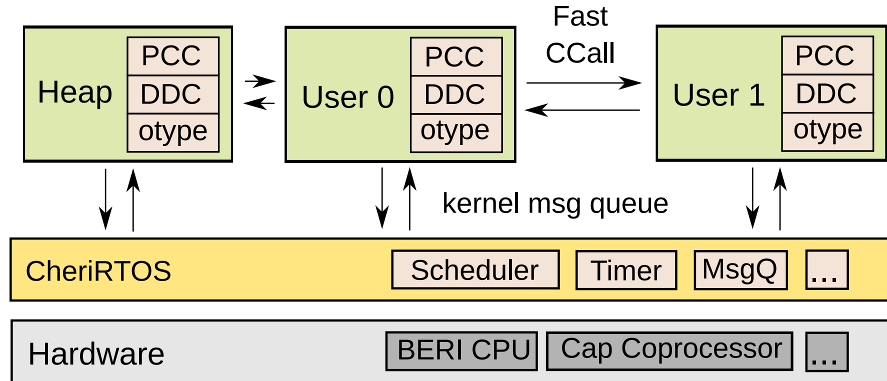


Figure 4.1: *Overall structure*

etc. Within this kernel, CHERI is the only deployed security mechanism. It aims to unify the necessary protections solely within one model. This implementation explores the possibilities while trying to maintain a low-latency and low-overhead profile. The following sections expand on the key design choices, which demonstrate how capabilities can fit nicely in such a model.

4.4.1 Overview

The structure of the CheriRTOS system is shown in Figure 4.1. At start-up time, the boot sequence retrieves the almighty capability (covering the entire address space with all permissions), derives a code and a data capability, and installs them. The almighty capability is then discarded. This fundamentally enforces $W \oplus X$ before any other capabilities are derived. CheriBSD preserves an almighty capability to support `rwx` pages, self-modifying code and capability retagging for memory swapping. However, none of these use cases are of any interest in an embedded system and CheriRTOS can afford to enforce this separation at the very beginning of the boot sequence, significantly complicating any effort to perform code injection attacks.

The boot sequence then loads the kernel and starts the initialisation sequence, which loads tasks from external storage to RAM. After copying its code and data to memory, a task is confined within a pair of code (PCC) and data (DDC) capabilities, with PCC restricted to its code section and DDC restricted to data sections and the stack. The closure of the PCC and DDC pair defines the initial domain of a task.

```
lw  $t0, 8($s0)       # implicit, check against DDC
                       # load addr = DDC.base+$s0+8
clw $t0, $s0, 8($c1) # explicit, check against $c1
                       # load addr = $c1.base+$s0+8
```

Figure 4.2: *Example of memory access instructions under CHERI. "$" denotes registers. Loading a word at address `$s0 + 8` (relative to the base of the capability) into `$t0`, either implicitly or via an explicit capability register.*

The Program Counter (PC) is allowed to fetch instructions only within the bounds and permissions of PCC. The system operates under the hybrid capability ABI, hence data loads and stores are restricted implicitly by the bounds and permissions of DDC by default. Any access outside one's domain can be granted only by receiving additional capabilities and explicitly specifying the capability to use instead of using one's own DDC (Figure 4.2). Note that this means by design, passing raw pointers between domains is inherently meaningless, as they can only be interpreted as offsets into the receiver's PCC or DDC but not into the sender's domain. The only option for memory sharing or communication is to send capabilities via capability APIs.

The overall structure reflects the requirement of an isolated and scalable system. All tasks reside within individual domains and the architecture does not impose a limit on the number of domains created. Theoretically, CheriRTOS scales up to arbitrary numbers of tasks, and should be limited in practice only by **otype**, memory and processing power.

4.4.2 OType space

The **otype** field steals 8 bits in total from **T** and **B** in a sealed CC-64 capability, which allows 256 different types. The kernel simply allocates one unique type for each task. 256 types is sufficient because it is impractical for the number of tasks in an embedded environment to exceed this limit. The type allocation is made by giving each task a "key" capability with the **permit_seal** permission, which has a length of only one and a unique base. This key allows the receiver to seal and unseal capabilities of the unique type. Upon receiving the key, each task now has the ability to create CCall handles and unforgeable tokens.

4.4.3 Non-PIC dynamic task loading

Following the specification of FreeRTOS, CheriRTOS implements dynamic task loading, albeit with a major difference and advantage. Dynamic tasks are tasks that can be loaded and unloaded during run-time. Since the load address of a binary cannot be known statically, it has to be compiled in a way that allows loading at arbitrary addresses. Without CHERI, two solutions exist: either patching all the addresses in the binary before loading it, or compiling it as Position-Independent Code (PIC). PIC adds a layer of indirection by treating symbol addresses as offsets to Global Offset Table (GOT) entries. Instead of patching all addresses, the binary loader can simply patch a few GOT entries to indirect all accesses around the actual load address, significantly reducing the complexity of run-time patching. However, it comes at a cost of extra indirections and memory accesses, which translates to a reduction in performance and an increase in power.

I take advantage of the fact that memory accesses in CHERI are offset by the base of capabilities. This offset property creates an obvious means to relocate binaries at near-zero cost. Figure 4.3 shows how a binary is patched to work at a different address.

In CheriRTOS, all code is compiled in non-PIC mode starting at address 0. To load it at a different location, I simply copy it to the target address and bound it with a PCC or DDC whose base is set to this address. The PCC and DDC will then offset any memory access, giving the task the illusion that memory still starts from 0 and that it has its own “address space”. As a matter of fact, supporting dynamic loading does not need any patching to the original binary, nor does it introduce any run-time costs, unlike the PIC solution.

4.4.4 Context switch

Capability registers have single-cycle access for most capability instructions. Context switches, exceptions, and interrupts will also store and load the capability register file (including PCC, DDC and the KR1C). This means a context switch also becomes a domain switch. Because capability registers can be stored or loaded like general purpose registers, we can efficiently maintain a capability context for each user task. Problems like secure interrupts, for example, are less of a concern because

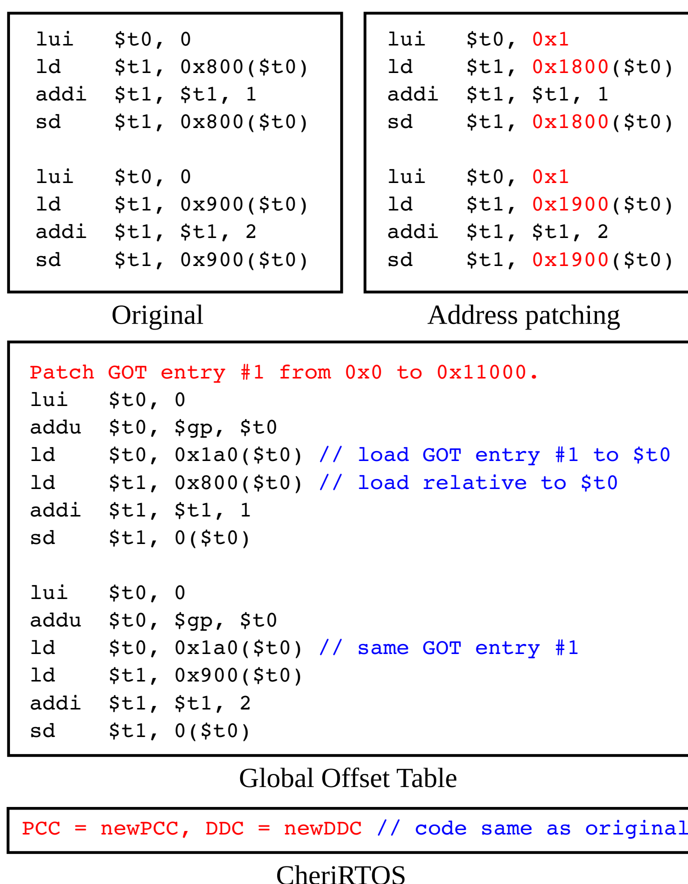


Figure 4.3: *Increment two variables at 0x800 and 0x900. Binary compiled for 0x0 now loaded at 0x11000. Red indicates patching.*

context switching to the interrupt code effectively switches to a new set of capability registers including PCC and DDC.

As capability register file switching is fast and efficient, unlike MPUs, CheriRTOS does not have to configure capability registers globally at boot time for all tasks, nor does it have an inherent limit on the number of tasks (and the number of protection regions for each task) that could be enabled simultaneously. Recall that RISC-V PMP needs 3 registers to describe a protection region, or 2 in power-of-two mode for double the number of regions. This at least triples or quadruples the register switching costs. Some ARM MPUs further divide each entry into sub-regions or implement regions as memory-mapped registers with much higher latency than a single cycle, which makes MPU context switches a huge overhead. Thus, commercial embedded chips with MPUs enabled never use them on a per-task basis. MPU registers are installed

on start-up to protect secret keys, kernel, code sections, etc., and remain largely static till the end.

4.4.5 CCallFast

The paper on fast domain crossing illustrates how CCall crosses and enables communication between domains with overheads on the same order of magnitude as function calls [67]. The CCall instruction in the paper is exception-based and the kernel services the exception like other system calls. Unfortunately, software argument validation, kernel entry/exit and hardware pipeline flushes still impose considerable overhead. The paper hypothesises a hardware-assisted CCall, where argument validation, register clearing, type matching and unsealing are done in dedicated instructions without exceptions.

CheriBSD, Lawrence Esswood's CheriOS implementation [27] and CheriRTOS all heavily rely on a fast exception-less mechanism for fine-grained compartments, making the hypothesised hardware assist necessary. Jonathan Woodruff implemented the first version of the CCallFast instruction. After that, many discussions were raised to improve and to finalise its semantics. I pick up the final specification and implement it in our CHERI hardware (Figure 4.4).

```
ccallfast $c1, $c2:
1   validate_args($c1, $c2)
2   PCC <- unsealed($c1)
3   DDC <- unsealed($c2)
```

Figure 4.4: *CCallFast sequence*

The `validate_args()` function performs a series of assertions to ensure the call is authorised in hardware, including sealing, permission checks and type matching. Upon a successful CCallFast, the sealed code and data capabilities are atomically unsealed and installed to PCC and DDC respectively, transferring control to the new domain. It is exception-less and allows atomic domain transitions within a ring. This instruction has a delay of only one cycle (the branch delay slot).

MIPS branch delay slots are a security vulnerability. I removed the CCallFast branch delay slot, making it the only jump/branch instruction in CHERI-

MIPS without a one-instruction delay. From the code above, CCallFast is similar to a jump-and-link instruction with the link content from an unsealed CCall data capability instead of PC+8. Initially, CCallFast followed MIPS by linking to the DDC register in the delay slot and jumping to the new PC in the next cycle. A direct exploit was immediately found by copying the unsealed DDC to another register in the delay slot. The caller then grabbed the unsealed capability of the callee for unauthorised access. The vulnerability was patched by disallowing any access to DDC in the CCallFast delay slot. However, Kyndylan Nienhuis pointed out that if an exception was fired in the delay slot, the unsealed DDC would be available after the exception exit for the CCallFast instruction, as the faulty Exception PC (EPC) of the delay slot was the branch itself. I attempted the exploit using the UNIX signal delivery mechanism to retrieve the callee's DDC. UNIX processes are able to register signal handlers to handle certain exceptions. The exception in the CCallFast delay slot is delivered to the caller's signal handler, and by examining the register file in the signal handler, the caller successfully reads the unsealed DDC of the callee. I have demonstrated this exploit on both QEMU-CHERI and the FPGA implementation. A quick mitigation was to clear the DDC in the kernel when a CCallFast delay slot generates an exception. To solve the problem, I modified the pipeline to implement CCallFast without an architectural delay slot. This fundamentally removed the vulnerability in hardware. Meanwhile, this incident brings attention to the interaction between MIPS delay slots and security. In the context of domain crossing, the design of MIPS' jump-and-link is inappropriate due to the fact that the CCallFast link register belongs to the callee and linking is performed before jumping. Since the delay slot instruction is still controlled by the caller, the linked register can be read or hijacked before entering the callee. This does not expose security vulnerabilities in the normal MIPS ISA, since domain switches do no occur on jump-and-link boundaries. However, when intra-address-space domain crossings happen with jump-and-link style instructions like CCallFast, we must guarantee that the pipeline does not expose any transient state between linking and jumping, or simply make sure no such transient state exists.

Direct inter-task communication via capabilities

FreeRTOS implements at least two APIs for inter-task communication. Firstly, message queues provides buffered communication between tasks. At the beginning of a communication, the kernel creates communication objects and buffers them in a queue.

Receivers do not receive the messages directly, but instead receive the intermediary objects dequeued by the kernel. Secondly, the task notification mechanism allows the sender to directly send one message to the receiver, circumventing any queue manipulation and temporary objects [4]. Apparently, queues permit buffered and more importantly non-blocking message sending at a cost of more kernel involvement, higher memory footprint and higher latency, whereas task notifications are a low-cost, low-latency mechanism with limitations.

In the baseline MIPS RTOS, an idle task waiting for messages will be woken up if the kernel finds outstanding messages in its queue. In CheriRTOS, as described in Section 4.4.4, context switching to the message receiver transitions to a new domain, which automatically guarantees isolation between two ends of the communication. However, the direct task notification circumvents kernel intervention. Without the exception path and the kernel, a user space mechanism must exist for atomic and isolated domain crossing.

CheriRTOS guarantees user space isolation in domain crossing using CCallFast and unique otype allocation. Each task is able to use the unique key to seal a pair of code and data capabilities as a handle, and distribute them to other tasks. This does not expose any memory to others due to the immutability and non-dereferenceability. The handles can only be used for CCallFast purposes. As described in Section 4.4.5, the atomic switch on PCC and DDC ensures the caller-callee isolation, which also means that any memory sharing must be done via passing capabilities instead of raw pointers.

Register safety

The domain of a task is defined by the transitive closure of all its reachable capabilities. CCallFast atomically replaces the PCC and DDC pair but not all registers. To avoid leaking capabilities or data, a safe CCallFast should back up all registers and clear them before the domain transition. Two CCallFast subroutines are provided in CheriRTOS. CCallFast API 1 assumes the callee can be trusted and only maintains caller-save registers. API 2 does not trust the callee to maintain the callee-save registers and therefore backs up all. The first API is intended for frequent calling of system tasks, e.g. the centralised heap manager or drivers, for further reduced latency. For

communication between two user tasks where mutual distrust is more appropriate, the API 2 should be used.

Capability "system calls"

The user space CCallFast enables another opportunity for low-latency. A user with limited privileges would often require the system to perform privileged operations, which is done via system calls. System call instructions serve no other purpose than generating an exception and signalling the kernel that a system call has been made. This is the common model adopted in almost all architectures today. Unfortunately, conflating system calls and the exception path is inherently high latency due to multiple CPU pipeline flushes and kernel handler routines. As a result, many researchers explore ideas of exception-less system calls [60, 59].

Switching domains with CCallFast requires no exception level transitions. CheriRTOS does this by using CCallFast heavily to provide "system calls" in user space like `timer_get()` and `tid_get()`. A task takes a sealed capability pair to the kernel to directly CCall kernel functions in user space, resulting in significantly lower latency. Of course, normal exceptions and interrupts still go through the exception path because they are unpredictable and are involuntary context switches.

Another advantage of CCallFast "system calls" is its interruptability. The CheriRTOS kernel is not preemptive, thus all kernel routines are designed to be short and deterministic to meet real-time requirements. For frequent system calls, the non-interruptable kernel routines may accumulate to be a significant portion of the task's run time, during which interrupts are blocked. In contrast, CCallFast system calls are normal user space routines that are fully interruptable. This also makes it less necessary to build a preemptive kernel.

Device driver isolation. Isolating device drivers is similar to capability sytem calls. I decentralise the kernel so that drivers are isolated into individual tasks. As a result, the kernel becomes smaller, further reducing the attack interface for a minimum Trusted Computing Base (TCB). For example, the UART module is just a task possessing a capability to the UART memory region. Whether another user task is able to access the UART is determined by whether it is granted the sealed capability pair to CCall into the UART task. Previously, `printf()`s are exception-based system calls into the kernel which has all the rights. Now, tasks which are allowed to call

`printf()` are performing low-latency CCalls into the UART task which only has access to the UART memory-mapped region. CheriRTOS not only restricts which tasks have access to a device, but confines the rights of the device drivers as well.

4.4.6 Return and real-time guarantees

It is crucial to enforce that a CCall to a potentially buggy or malicious callee returns and meets the deadline. However, a direct CCallFast cannot guarantee this on its own. To return from a CCall, the caller's PCC and DDC have to be securely restored. I implement distributed trusted stacks to protect these capabilities from tampering by the callee, as shown in Figure 4.5.

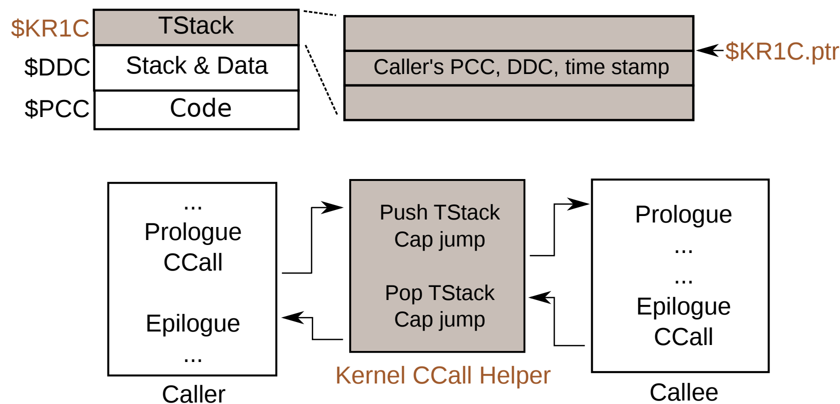


Figure 4.5: *Trusted stack and CCallFast round trip. Dark indicates kernel-only objects. The pointer field of $KR1C points to the top of the trusted stack.*

Upon task creation, the memory allocator also allocates a small trusted stack for each task. A kernel capability register ($KR1C, not accessible to user tasks) is reserved to point to the trusted stack, and a kernel CCall helper is inserted between the caller and the callee. With a trusted stack, a CCallFast first calls into a kernel helper, which pushes the caller's PCC and DDC onto the trusted stack, and then calls into the callee. A CCall return also returns into the helper, popping the caller's PCC and DDC before jumping back. Note that the centralised CCall helper also changes how sealed capabilities are distributed. Before, sealed capability pairs are directly distributed to the callers. Now, a sealed pair is submitted to the helper, and another sealed capability is created by the helper as an identifier to the actual pair. Potential callers only receive the sealed capabilities to the CCall helper and the identifier. The helper maintains the trusted stack and does the actual CCall on the caller's behalf.

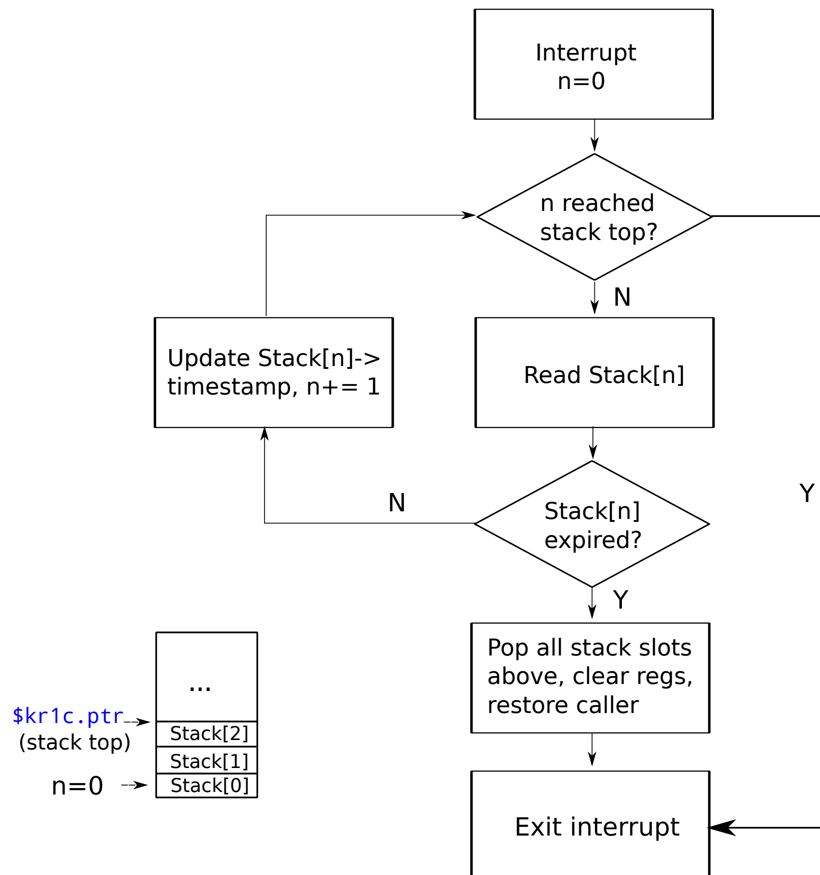


Figure 4.6: *Interrupt routine to check for expired CCalls*

These measures ensure the caller's return information cannot be tampered with or cleared; as a result, it is always possible to securely return to the caller regardless of the callee, further enforcing the control flow between domains.

In addition, a caller may also specify a timeout for a CCall, which will cause a time stamp to be recorded in a trusted stack entry. Time stamps are regularly checked by the timer interrupt, and force the callee to return after expiry. The trusted stack is traversed from the bottom to the top but not vice versa (Figure 4.6). The reason is that if A calls B with a time stamp of x and then B calls C with a time stamp of y, when x expires, all subroutines called by B should also be considered expired, therefore there is no need to scan the stack any further, whereas scanning from top to bottom does not have this simplicity. Popping the stack restores the caller's context,

and assigns zeros to return registers to signal the caller that the CCall has expired and real-time mitigation action may be required.

A caveat found in evaluation of the real-time guarantee is that the real-time check routine itself is not deterministic. The time spent to scan the trusted stack may vary due to different depths of CCalls, so the interrupt latency can become unpredictable. To compensate, the size of the trusted stack should be limited, both to increase determinism and to prevent malicious callees from creating a very deep stack (e.g. CCalling itself recursively) that severely disrupts real-time checking.

Note that Figure 4.5 omits CCall helper's interaction with the scheduler, assuming no scheduling events in the round trip. To guarantee real-timeness in practice, the return path of the CCall helper also calls into the scheduler to check work queues, ready status and task priorities to determine whether a re-scheduling is required. For example, a CCall from a low-priority task that manipulated a device might have also received input for a high-priority or real-time task. Under such circumstances, the scheduler re-schedules for a high-priority task and performs a context switch. The CCall return will be resumed after the completion of high-priority tasks, thus the round trip may not be as straightforward as the diagram depicts.

4.4.7 Secure centralised heap management

Section 2.4.1 explains why a centralised heap is a common source of memory safety vulnerabilities due to shared and unbounded access. Tasks that request heap allocations usually have access to the whole heap and its data structures. Worse, the heap shares the address space with all tasks and the kernel, leading to potential escalations of memory safety attacks like in the Broadcom WiFi example.

Properties of CHERI capabilities – fast domain crossing, fine-grained memory protection and capability unforgeability – provide the excellent infrastructure for solving the problem of a centralised shared heap. Note that in Figure 4.1, the heap manager is no different from a normal user task to the rest of the system. The only significance is that it has a relatively large initial domain that covers the whole heap. Internally, the heap adopts `dlmalloc()` style metadata and "free-lists" for maintenance (Figure 4.7).

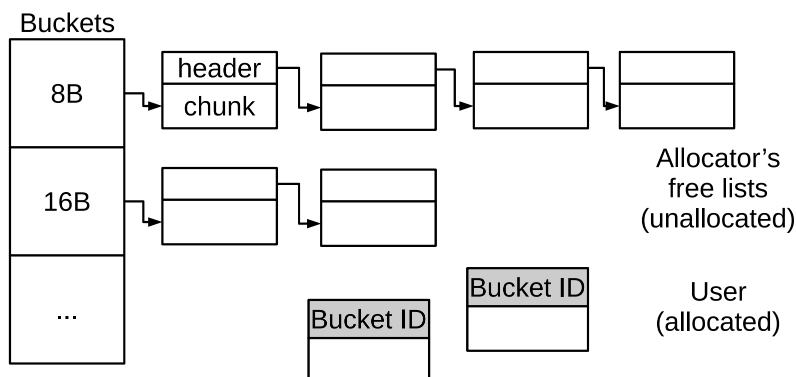


Figure 4.7: *Memory allocator structure (gray boxes indicate that the bucket ID is sealed inside a sealed capability)*

Overall, the allocator maintains multiple *bins* for multiple sizes of free memory chunks. If multiple chunks belong to the same bin, they are chained into a linked list, which means the header of a free chunk contains a capability to the next in the bin. A request of size n larger than the size of bin $m-1$ and not larger than the next bin m will pick bin m and unlink the first chunk. The unlinked chunk no longer belongs to the bin and its header is modified to a sealed capability with the heap manager's otype. To remember which bin to return to upon `free()`, the offset in the sealed capability is an integer of the bin index. The unlinked chunk is restricted by a CSetBounds to exclude the header and to restrict it to n, before being returned to the caller via a CCallFast return. Upon `free()`, the capability has to be rederived to include the header. The heap manager unseals the header to know which bin it belongs to. Then, it needs to rederive and grow the capability to its original size based on the bin index. The heap has to store the bin index instead of using the length of the capability to derive the bin, because the user might have performed CSetBounds on it and the heap may then put it in a bin of smaller size, losing available memory.

The advantage of this design is twofold. First, as pointers are meaningless to another task, heap memory has to be accessed via bounded capabilities, protecting other allocations and the heap itself. Section 2.4.1 points out the alternative approach to separate chunk metadata from the chunk itself to avoid metadata contamination. However, bounded access via capabilities makes this design unnecessary, and it is safe to use inline metadata for low-latency. Second, the unforgeability and the unique otype guarantee the correct flow of memory chunks. It is impossible for a user task

to free arbitrary memory outside the heap, because the sealed capability header will not be present and the `free()` call will be rejected. The double free problem is also impossible. The first `free()` call mutates the sealed header back into a linked-list capability. The second call will therefore fail to see a valid sealed token and will be rejected. By incorporating the unique sealed token, it is guaranteed that capabilities flowing between the user and the heap manager originate from the heap region, and the free-lists are in valid states.

4.5 Evaluation

4.5.1 The MiBench benchmark suite

For the evaluation section, proper benchmarks need to be selected to mimic the typical workload on IoT devices. Common benchmark suites, e.g, SPEC-CPU, Dhrystone, Olden and Octane, are not suitable for this task as they target larger operating systems which include support for multi-threading, virtual memory, file system handling, etc., and focus largely on overall CPU performance but little on latency and real-time behaviour. For these reasons, the benchmark suite MiBench was chosen. MiBench is a free, commercially representative embedded benchmark suite that shows considerably different characteristics than SPEC2000 when analysing the static and dynamic instructions of embedded processor performance. It is composed of various benchmark categories including automotive and industrial control, network, security, consumer devices, office automation and telecommunications [32]. These benchmarks attempt to replicate typical use cases which are closer to IoT scenarios than existing CPU performance-oriented benchmarks.

Capabilities allow benchmarks to run within their own sandboxes, making the organisation of them inside CheriRTOS relatively straightforward. After the implementation of the kernel, I also port a large number of benchmarks to CheriRTOS, including qsort, AES, stringsearch, dijkstra, spam, sha, CRC32, bitcount and adpcm. The porting effort is mostly adopting the benchmarks to the task isolation and communication primitives of CheriRTOS, converting system calls into capability-based CCalls and reorganising memory allocation. The benchmarks ported can be partitioned into three categories:

1. Self-contained benchmarks: **qsort, stringsearch, bitcount, adpcm, CRC32**. These benchmarks involve data processing on certain datasets and function within their own compartments.

2. Domain crossing: **AES, SHA**. These benchmarks are converted into CheriRTOS core services which operate in separate compartments. A user module, **CCallTest**, performs encryption and decryption, hash computing via domain crossing into these services repeatedly.

3. Memory allocation: **dijkstra**. This benchmark generates large graphs to compute minimum distances, requiring frequent dynamic memory allocations and frees.

4.5.2 Non-PIC and compartmentalisation

The immediate advantage that capabilities bring is the ability to create isolated tasks and to "offset address spaces". As explained in Section 4.4.3, PIC is no longer compulsory, which immediately gives a performance advantage during run-time due to fewer instructions and fewer memory indirections. Comparing non-PIC with PIC binaries requires proper loading of both. For evaluation purposes, the kernel is augmented with PIC loading routines which identify ELF headers and sections to rewrite GOT entries around actual load addresses, whereas non-PIC loading is mostly just a `memcpy()`.

All ported benchmarks are compiled both in non-PIC and PIC. Below demonstrates the performance. Total cycles, number of instructions and memory access statistics are reported (Table 4.1).

It is clearly seen that non-PIC alone is able to contribute to a performance benefit. Although less drastic than the high variation on x86 [52], the impact is still ranging between 0% and as high as 15%, which is not negligible. The PIC indirections are reflected by the number of instructions and the memory access behaviour. Table 4.1 shows that the additional data traffic can be as high as 56% (35.93% reduction) for bitcount.

Two benchmarks that show the most overhead in PIC are qsort and bitcount. Both have a rather modular design by splitting the code into small functions, and

call them repeatedly from multiple call sites. This behaviour suffers significantly from additional PIC function call prologues. For non-capability systems in which PIC is compulsory, these benchmarks should merge and explicitly inline tiny functions where appropriate in order to mitigate the cost.

The experiment also discovered a compiler defect, which explains why in few cases, non-PIC performance is worse. MIPS PIC reserves **$gp** for a pointer into GOT, and further accesses to GOT can simply issue a load based on **$gp**. However, the non-PIC mode of the compiler synthesises the address each time before loading a global variable, which clearly can be optimised by temporarily reserving a register for continuous global access.

4.5.3 Fast and direct domain crossing

With `CCallFast`, we are able to bypass the kernel and exception paths, and directly CCall into the callee – significantly reducing the latency of inter-task communication.

The results show the decomposition of instructions and cycles respectively in Figure 4.8. The benchmark is done by sending a short message from a user task to an encryption task, and returning immediately. The first item is a direct service call in the MIPS setup, which offers absolutely no protection and acts as a baseline, and strictly speaking not a CCall. Under CheriRTOS, a jump with a raw pointer is impossible due to task isolation, and must be done via CCall. I implement two CCall paths: one signals an exception to a kernel handler that performs software otype checking and unsealing before jumping to the new domain, similar to the exception-based domain

	Code	qsort	AES	SHA	CRC32	string	dijkstra	adpcm	bitcount
Cycles	PIC	0.875	3.501	1.897	3.258	0.631	0.837	0.909	1.305
	non-PIC	0.807	3.456	1.908	3.257	0.601	0.746	0.899	1.204
	reduction	7.77%	1.29%	-0.58%	<0.01%	4.75%	10.87%	1.10%	7.74%
Instr.	PIC	0.611	2.923	1.770	2.635	0.546	0.616	0.805	1.358
	non-PIC	0.555	2.894	1.768	2.635	0.529	0.590	0.819	1.204
	reduction	9.17%	0.99%	0.11%	<0.01%	3.11%	4.22%	-1.74%	11.34%
Mem.	PIC	0.183	0.567	0.353	0.154	0.143	0.165	0.526	0.176
	non-PIC	0.162	0.546	0.351	0.154	0.136	0.130	0.525	0.113
	reduction	11.48%	3.70%	3.36%	<0.01%	5.40%	21.64%	0.15%	35.93%

Table 4.1: *MiBench: non-PIC vs. PIC (all numbers in billions)*

crossing in Cortex A and R TrustZone; the other is the fast CCall path, which uses the `CCallFast` instruction to perform hardware and exception-less inter-task calls. Some steps need further clarification. The caller's prologue involves setting up arguments and CCall capabilities. The called service acquires a mutex before entering the critical section due to CheriRTOS supporting multi-tasking. After the service call is finished, it releases the mutex and returns.

For the capability domain crossing, even without packing multiple checks and unsealing into the CCallFast instruction, removing four pipeline flushes has already contributed to 40 cycles, which is 26% of the total round trip. With the atomic checks and unsealing, jump based CCall removes another 25 cycles (15%). Taking all changes into account, a reduction of 64 cycles is achieved in total, improving the CPI drastically from 1.64 to 1.17. This is the result from a version of CheriRTOS that already supports multiple services in a domain and multitasking. For even lighter use cases where only a single thread is running at a time, service number checking and mutex handling can be removed. An experimental single-thread configuration shows a round trip of only 63 cycles for jump based CCalls, whereas the exception path takes 127 cycles, a CPI of over 2.0.

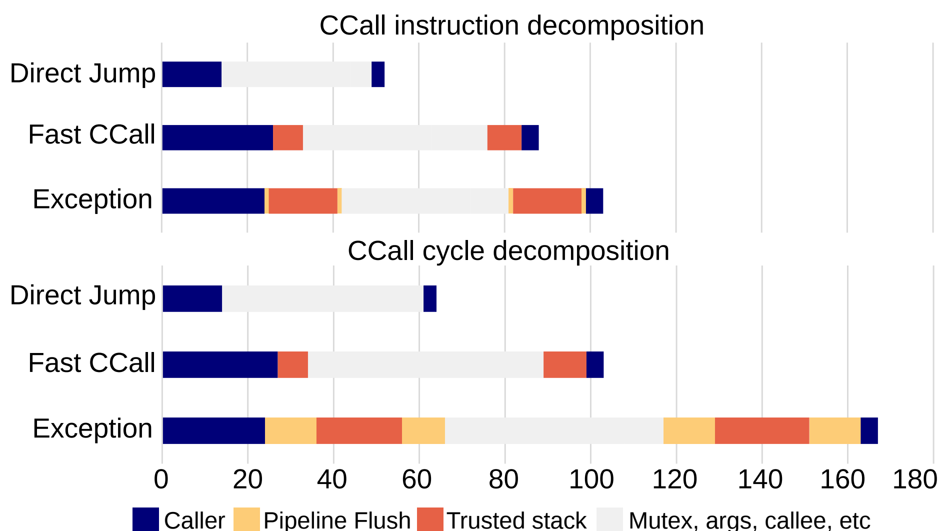


Figure 4.8: *Instruction and cycle counts for a round trip: direct jump vs. capability jump (fast CCall) vs. exception based CCalls*

4.5.4 Register safety, return and real-time guarantees

The previous evaluation focuses on basic memory safety and task isolation, which is only sufficient when the caller has certain level of trust of the callee. Obviously, there is a spectrum of protection and trust levels that need to be explored, shown in Table 4.2. These protection levels have all been explained in Section 4.4.5.

I extend the CCallFast routines with the additional guarantees and evaluate their overhead in Figure 4.9. The bar chart at the top is additive, and the next protection level includes the costs from the previous level. The bottom shows the additional costs in the timer interrupt. Both the return and the real-time guarantees require the timer interrupt to check the time stamps on the trusted stack. The return guarantee computes a hash and examines a random stack slot at a time, while real-time guarantee has a cost that depends on the run-time trusted stack depth, between examining only one stack slot (min) and examining a full stack (max).

The full traversal inspects all slots on the stack and checks expiry for all time stamps. This full traversal adds non-determinism, as the timer interrupt delay now depends on the depth of the trusted stack, but may be necessary for tasks requiring precise real-time guarantees. The worst case performs a full trusted stack traversal which can add a maximum delay of 100 cycles when all 4 stack slots are visited, and creates a variation of 60 cycles (0.6μs) between the extremes. The non-determinism of the timer interrupt delay is the very reason why the depth of the trusted stack is restricted to only 4. The depth could be limited even further for more determinism. In practice, a depth of 4 might still be too permissive. The deepest call chain observed in this benchmark setup is a round trip of an encryption cycle from a user, in which the chain involves the user calling the AES module, which calls the memory allocator

Protection level	Overhead
Simple CCallFast	Setting up capability arguments for trampoline and callee
+ register safety	Clear registers. Save and restore callee-save registers
+ return guarantee	Additional flags and random checking of the trusted stack
+ real-time return	Full trusted stack traversal

Table 4.2: *Spectrum of protection levels and overhead*

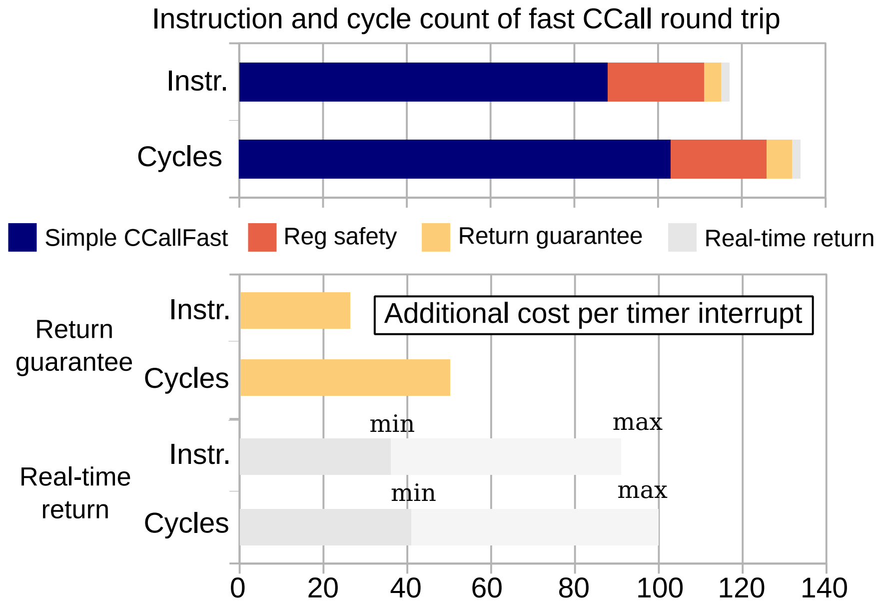


Figure 4.9: *Overhead of different protection levels*

for a temporary buffer. The allocator under debug mode may call the system logger to write debug messages into a circular trace buffer, which later (after the real-time task ends) emits output to UART or can be read for offline analysis. This example creates a CCall depth of 3. As for almost all cases, a CCall depth of 1 or 2 may already be sufficient, as it is unlikely that a real-time task would perform a `CCallFast` to further deepen the stack.

4.5.5 Overall system performance

This section investigates the impact of the overall system performance by comparing the CheriRTOS implementation with its pure MIPS baseline. The pure MIPS kernel contains all modules in a single physical address space. Self-contained benchmarks access their datasets (e.g., arrays to be sorted in qsort, strings to be searched in stringsearch, etc.) on the heap outside the domains of the benchmarks via constrained capabilities. For the rest of the benchmarks, AES and SHA stress domain crossing performance and dijkstra stresses both domain crossing and heap allocation. Another user task, *ccalltest*, CCalls into AES and SHA to perform encryption and message digest respectively on 1MiB of data. Due to the tight memory budget, the test has

a data buffer of only 8KiB, therefore 128 domain crossings are required for AES or SHA. In the CHERI case, these are safe domain transitions with memory safety; in the baseline case, these are simple function calls passing unprotected pointers.

However, directly comparing the two may result in an unfair advantage towards CheriRTOS due to non-PIC alone contributing to up to 15% of the performance increase. To expose the overall cost of a capability system, all benchmarks under CheriRTOS are compiled into PIC (although unnecessary) to give up this advantage. All benchmarks now should show the overhead of capabilities.

I set the timer interrupt at 100Hz, the suggested rate by FreeRTOS, to also detect the expiry of real-time tasks (a resolution of 0.01s). The results are shown in Figure 4.10.

Overall, the cycle overhead falls below 5%, varying from 4.7% to almost no overhead. Dijkstra sees the highest overhead because each node of the graph is dynamically allocated with CCall. In addition, as a capability is double the size of a 32-bit pointer, graphs constructed with capabilities have a larger cache and memory footprint. Despite these issues, Dijkstra's cycle count is still only 4.7% above the insecure baseline.

In two benchmarks, namely AES and adpcm, a negative overhead is sometimes observed. Tracing shows that having an additional 8 registers used for capabilities relieves register pressure and reduces stack loads and stores. Although in the extreme

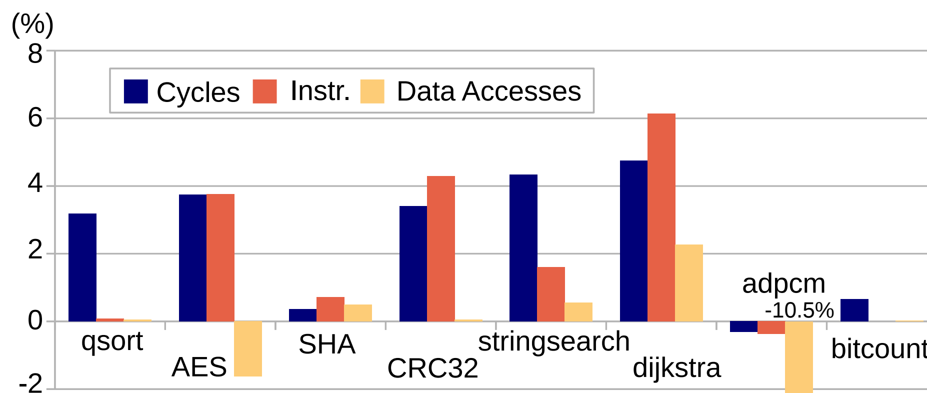


Figure 4.10: *Overall overhead across benchmarks*

case data accesses are reduced by 10%, they are typically on the stack with good spatial locality, which normally hit in the data cache and have less impact on cycles.

4.6 CheriRTOS vs. state-of-the-art

This section highlights how CHERI provides a superior substrate to the state-of-the-art solutions, especially in terms of meeting the goals set in Chapter 2.

Task isolation. CheriRTOS is able to create an arbitrary number of mutually isolated tasks[1]. In contrast, MPU and TrustZone have trouble supporting more simultaneous tasks than the number of protection regions the scheme supports.

Fine-grained memory protection. Due to the limited number of protected regions, state-of-the-art solutions cannot protect memory at arbitrary granularities or on a per-object basis. CHERI capabilities are user space types that can flow between registers and memory just like traditional pointers, having no restriction on the number or the size of protected objects.

Fast and secure domain crossing and inter-task communication. The memory protection components in state-of-the-art solutions are normally kernel-only, thus switching between domains is inherently impossible without kernel intervention. CheriRTOS offers direct inter-task communication via `CCallFast`, which bypasses the kernel and switches the domain atomically, purely in user space.

Secure centralised heap management. Secure and performant centralised heap management depends on fine-grained memory protection, task isolation and fast inter-task communication. Here, the prerequisites are all met.

Real-time guarantees. Non-CHERI commercial RTOSes already provide real-time primitives like real-time events and timers. CheriRTOS makes them secure by forcing a domain switch on each context switch and by implementing the trusted stacks. Malicious callees are unable to block the system, and the real-time handler's or the caller's context cannot be tampered with.

[1] Apparently, a unique otype per task restricts the maximal number of tasks. CHERI-64 supports 256 otypes which is much higher than typical embedded processors can multitask. Even if the otype limit is a concern, alternative designs exist like allocating a single otype for all user tasks and using additional identifiers to distinguish among them.

Scalability. As described previously, there are no restrictions on the size and the number of tasks and protected regions. Having multiple isolated tasks each accessing the heap via numerous protected capabilities is simply not achievable via state-of-the-art solutions.

4.7 Summary

In this chapter, I presented CheriRTOS, a CHERI-based real-time operating system for embedded systems. A capability approach addressed many existing memory safety problems for embedded systems, and also enabled novel, efficient, and scalable solutions to task isolation with control-flow robustness. The evaluation confirmed that these benefits could be achieved without violating the performance and determinism constraints of embedded systems.

The review of state-of-the-art security architectures for embedded systems in Chapter 2 demonstrated the difficulty in finding a comprehensive security framework that is efficient, scalable and generic. Nevertheless, this implementation relied on the fundamental protection mechanisms of CHERI, which were utilised by the CheriRTOS platform to enforce fine-grained memory protection, task separation with fast and secure inter-task domain crossing, secure device drivers, secure centralised heap allocation, return and real-time guarantees.

I envisage a future where a fine-grained and unified security interface eases the safe design and deployment of embedded software systems. For example, removing position-independent code or run-time relocation results in a much simpler binary loader; separating peripherals from the kernel reduces its complexity and the attack surface; fine-grained memory protection removes many manual (and likely incomplete) bound checks and assertions made by programmers, resulting in higher assurance and performance. New software stacks targeting CheriRTOS specifically could easily achieve high robustness and efficiency, with less effort than state-of-the-art approaches.

4.7.1 The problem of temporal memory safety

Capabilities preserve bounds and permissions but do not prevent temporal memory attacks. If carefully designed and timed, an attacker could return a memory chunk to the heap manager by calling `free()` but still retain the capability. Later, the memory chunk is allocated to another user which now the attacker has access to. So far, no special measures are implemented to defend against such attacks.

The next chapter will investigate how CHERI can take advantage of its spatial safety properties to defend against temporal memory vulnerabilities.

Chapter 5

Temporal safety under the CHERI architecture

A question that arises from the previous chapter is how to prevent a centralised and dynamic heap from being exploited temporally. The difficulty is tied to the nature of CHERI capabilities: they are unforgeable tokens designed to enforce bounded memory access, but do not possess any temporal semantics. However, capabilities being bounded and distinguishable from integers is sufficient to make temporal safety possible. In this chapter, I explore how feasible temporal memory safety is for CHERI, proposing and implementing architectural, micro-architectural changes and software prototypes, especially for the sweeping revocation scheme. As temporal memory safety under CHERI is a relatively new frontier, this chapter does not restrict its scope to the embedded space, but focuses on the most common denominator of both the desktop and the embedded configurations. The results, conclusions and research byproducts, I think, are applicable to a variety of CHERI situations and should be the prerequisites of more elaborate and complex CHERI temporal safety schemes in the future.

5.1 Background

This section gives a quick overview of the defenses and mitigation techniques against temporal memory safety in particular.

Software defenses

Temporal memory safety violations have been one of the major sources of security vulnerabilities for unsafe languages like C and C++. Due to the lack of bounds and time information in such languages, guaranteeing temporal safety can be as challenging as enforcing spatial memory safety. FreeGuard already offers an overview on the categories of heap memory safety violations as well as existing debugging and defense tools [58].

Apart from guarding against spatial overflows, some memory allocators directly target temporal safety by building safe-reuse allocators. The Cling allocator [1] proposes a type-safe strategy, where allocations of the same type are grouped within the same buckets and memory reuse cannot occur across buckets. Since most use-after-free attacks hijack function tables of a different type, this defense works reasonably well. However, the functionality depends on identifying objects of the same type, which many require examining the source program, identifying the call sites or even unwinding the stack. Failing to correctly extract type information leads to ineffective reuse separation and possible vulnerabilities. FreeGuard [58] combines multiple defenses and optimises them for production use, in which one idea used by temporally-safe allocators is to buffer freed chunks in quarantine zones for delayed reuse. To actually invalidate access to freed memory, allocators also use paged memory to implement virtual address non-reuse and to trap illegal access [51, 19]. Of course, using paged memory this way has its shortcomings, like the coarse granularity, the large sizes of page tables and the reduction in TLB efficiency. For debugging, AddressSanitizer creates auxiliary bit maps to track allocated spaces [56]. A mismatch between the bit map and a memory access signals an attempt to visit freed memory. However, such debugging tools usually come at high run-time costs and are cautiously used in production.

Hardware support

In addition to the attempts of developing temporally-secure allocators and debugging facilities, other schemes attack the problem directly at the hardware and architectural level.

Historically, many centralised capability systems store capabilities in dedicated data structures, and applications possess tokens that reference actual capabilities in

capability tables [16, 73]. A memory access is done via an indirection through the table. To remove dangling capabilities, a *distributed* capability system like CHERI must sweep all reachable memory to locate all capabilities a process could have created, whereas a centralised table takes significantly less effort to locate and invalidate capabilities. However, these systems incur a high overhead due to the cost of capability indirections and lookups among other performance penalties, and are not widely adopted in production.

Another category of hardware temporal safety involves memory tagging. Memory tags can assist at different levels and different granularities, annotating address validity, version numbers, object types, ownerships and so forth [31]. For example, the SPARC Silicon Secured Memory (SSM) tags pointers and cache lines with version numbers. An access via a marked pointer must also match the version of the cache lines, otherwise an exception is raised [50]. This enables the invalidation of stale references by simply changing the version numbers in cache lines, although problems arise when we run out of version numbers and are forced to reuse them, and it is yet to be seen how much stronger the security guarantee is when systems are built around such a mechanism. Similarly, the AArch64 HWASAN combines memory tagging (using top-byte-ignore) with a modified compiler toolchain for a hardware-assisted AddressSanitizer-like scheme [56, 57], which is also using unused bits in pointers as memory tags to detect stale references.

As previously discussed, many temporally-secure allocators are built by limiting the reuse of virtual addresses and leveraging the MMU. Similarly in peripheral accesses with I/O virtual addresses, the Input-Output MMU (IO-MMU) not only translates addresses but also blocks any stale references from peripherals. This is done by quickly unmapping IO-MMU entries after a transaction with a peripheral is finished, and assigning new mappings for later transactions [53]. However, as MMUs were not originally designed for temporal memory safety, these schemes typically offer coarse-grained solutions only at the page level. Meanwhile, poor TLB performance and the overhead from frequent map and unmap calls is the major factor why dynamic IO-MMU mapping is not enabled in most systems [53].

Probabilistic in nature. Note that till this point, no scheme guarantees deterministic temporal memory safety. Any scheme is forced to reuse a resource at some point of program execution. Quarantine buffers, for example, eventually fill up and

cannot grow any further; address non-reuse cannot grow the virtual address space indefinitely due to page table sizes and TLB inefficiency; the limit on the number of colours for memory tags means they eventually wrap around. Once a resource needs to be reused, it no longer prevents old references from being dereferenced. Thus, the aforementioned schemes usually present probabilities: memory tagging with n tag bits has a probability of $\frac{1}{2^n}$ to detect tag mismatch on average; Cling type identification relies on heuristics of the call stack and there is the probability of failing. In practice, the detection or mitigation probabilities of said schemes are considered sufficiently high, but this means attacks are still possible. Also, in the hands of an attacker, an average probability might not be useful. Take ASLR as an example, its randomness can be dramatically reduced by attacker's intervention and active probing, like in [30]. In the end, a probabilistic approach guards against bugs but may not defend well against dedicated attacks.

5.2 Opportunities of CHERI temporal safety

CHERI does not target temporal memory safety directly, and a valid capability is valid at any point on the time axis. However, the most notorious form of temporal violation, use-after-reuse, can still be prevented. Note that I use *use-after-reuse* to phrase the problem because it is more accurate. A pure use-after-free without reallocation of the underlying memory is less dangerous, which usually only results in heap metadata corruption that can be prevented by metadata segregation. In practice, it is almost guaranteed that a use-after-free will eventually become use-after-reuse, as a reasonable memory allocator will try to buffer freed memory chunks and hand them back to the user for efficiency. However, the difference between the two is important because the temporally-safe allocator later in this chapter enforces temporal safety but does not prevent any dereference on freed (but not reused) memory.

5.2.1 Deterministic temporal memory safety

Notably, CHERI achieves non-probabilistic safety through its two properties: unforgeability and monotonicity. If a heap allocator distributes a capability to user code, it is guaranteed that only subset capabilities will ever exist in the user. To disable access to a memory region, one can sweep through the address space and

identify capabilities pointing to freed memory. Several approaches exist to identify such capabilities, including shadow maps and subset testing (explained later). Once a stale capability is found, one may either zero the capability or strip the tag to revoke its rights.

Revocation, the act of *retracting* granted authority, is a key design choice in any capability system. In a centralised capability system, a capability table is the single accessible point for revocation state with strong integrity as described in Section 5.1. In other systems like CHERI, capabilities are distributed throughout the system, protected via other means, and must be all swept and deleted in order for revocation to take place. This dissertation uses the same term *revocation* for distributed capability systems as well to align with existing literature, although the details differ substantially.

A *revocation pass* or *sweeping pass* in CHERI invalidates all stale references pointing to certain freed memory, which can now be safely reallocated to other consumers, knowing that no stale references could exist at this point. To the best of my knowledge, sweeping is necessary for revocation in distributed capability machines.

The properties of CHERI let us precisely find and revoke all stale references, giving deterministic temporal memory safety. In contrast, conventional architectures cannot support the concept of revocation; once the underlying memory is reallocated, the defense on a conventional machine immediately becomes probabilistic at best.

5.2.2 Spatial and temporal safety combined

Unlike other aforementioned protections, which usually handle one particular category of vulnerabilities at a time, CHERI deterministically and architecturally guarantees both spatial and temporal memory safety. In fact, the CHERI temporal safety scheme described so far depends on the strong spatial safety provided by bounded capabilities: without bounded memory access, it is impossible to identify capabilities pointing to a certain memory region, thus revocation as a temporal defense is also impossible. In other words, CHERI temporal memory safety co-exists with its spatial safety, giving much stronger guarantees than previous partial and probabilistic approaches.

5.2.3 Possible but inefficient

The theoretical possibility of deterministic temporal safety does not translate well into a practical implementation if the CHERI ISA remains in its current form.

Locating capabilities. Sweeping through memory to check whether each word is a capability and whether a capability needs to be revoked is costly. To identify whether a memory word is a capability, one must load it into a register before performing `CGetTag` to get the result. This means all memory words are fetched into the cache, even though most of them do not have their tags set and are ignored immediately, wasting significant amount of DRAM traffic and cache capacity. To accelerate, one should be able to quickly locate only valid capabilities to avoid unnecessary work. This is significant for pointer-light applications, since such a mechanism allows the sweeper to skip pure data regions to dramatically reduce the overhead of sweeping.

Lack of temporal-safety-aware software for CHERI. Current software stacks ported to CHERI do not have a focus on temporal safety. For example, heap memory allocators for most modern OSes (like `dlmalloc()` and `jemalloc()`) prioritise the caching of freed memory chunks. They attempt to reuse chunks as much as possible for minimum memory footprint and maximum cache efficiency. Of course, this policy directly conflicts with temporal safety and will not function well with revocation. For example, to re-allocate a memory region to a `malloc()` call, stale references to this region need to be invalidated first. Unfortunately, under a maximum reuse policy, this could potentially require a memory sweep for each `malloc()` to perform invalidation, introducing unacceptable overhead. As a result, software stacks must be redesigned to be aware of temporal safety semantics.

5.3 Optimising for efficient sweeping revocation

The cost of sweeping revocation depends on two major factors:

$$Overhead = cost\ per\ sweep \times frequency\ of\ sweeping$$

Improving the efficiency should focus both on reducing the cost of each sweep and on minimising the frequency of sweeping in an application. Therefore, the two themes after this section are:

- Hardware assists to rapidly locate and revoke capabilities through memory, which reduce the cost of the first factor.
- Design of a new memory allocator to minimise the second factor.

5.4 Architectural/microarchitectural proposals and implementations for fast sweeping

I implement two important optimisations that dramatically increase the efficiency and overall speed of memory sweeping as well as reducing the corresponding DRAM traffic. Both are implemented in the CC-64 setup.

- `CLoadTags` to read multiple tag bits from the tag cache with non-temporal semantics.
- Page table *cap-dirty* bits to locate capabilities on a page granularity.

5.4.1 CLoadTags

A tagged architecture in CHERI extends each memory word with a tag, which requires all data paths (registers, caches and DRAM) to handle addtional tag bits. In the current implementation, tags in the memory hierarchy are depicted in Figure 5.1.

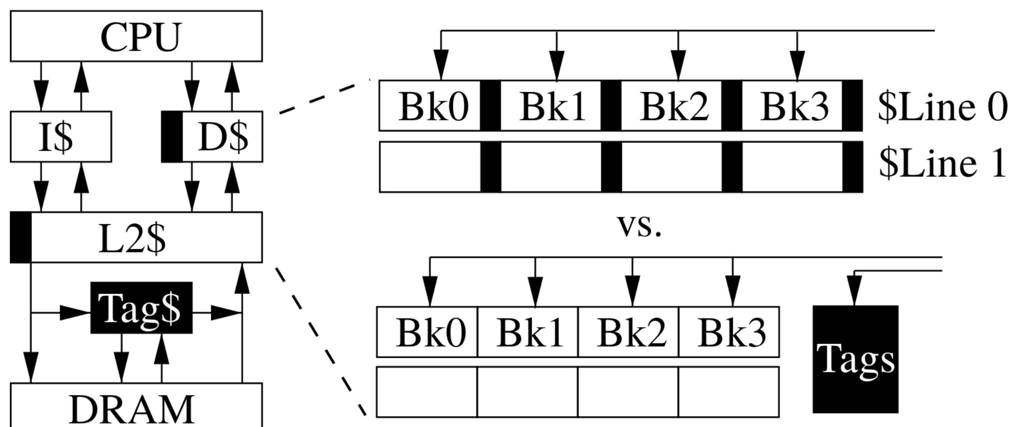
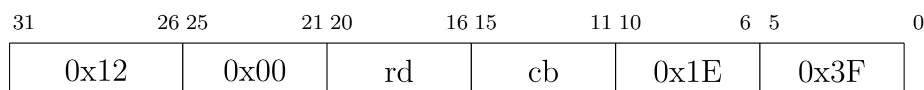


Figure 5.1: *Tags in the CHERI memory hierarchy and the refactoring of caches*

Chapter 2 already describes how CHERI uses off-the-shelf DRAM to support tags and how a tag cache improves its efficiency.

I modify the pipeline and the L1 and L2 caches to enable fast tag access. Before, tags in each cache line are interleaved into 4 banks with data, thus reading all tags in a cache line requires 4 read accesses. I separate data and tags, which means each capability access now issues two separate lookups (Figure 5.1 right). In this implementation, it has minimum impact on the critical path as I parallelise the two lookups. With this structure, it enables the hardware to do bulk tag reads (far more than the number of tags in a single cache line bank) in a single access. Further, I modify the tag controller to accept tag-only requests. Before, each request that reaches the tag controller will issue a lookup in the tag cache and a request to DRAM. The tag controller then combines the two responses to return actual tagged memory words. When a tag-only request is received, the controller does not visit DRAM, but only returns the tags from the tag cache as data, and the number of tags returned can be up to a register width (64 bits under CHERI-MIPS).

Format: CLoadTags $rd, $cb



Semantics

```
let vAddr   = getCapCursor(cb_val);
let vAddr64 = to_bits(64, getCapCursor(cb_val));
if (vAddr + 16 * cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
else if not (vAddr64[6..0] == 0b0000000) then // 128-byte cache line
    SignalExceptionBadAddr(AdEL, vAddr64)
else
    let pAddr = TLBTranslate(vAddr64, LoadData);
    for i in [0..16)
      let tag[i] = MEMr_tag(pAddr + i*cap_size);
    wGPR(rd) = zero_extend (tag[]);
```

Figure 5.2: *CLoadTags. Assuming 128-byte cache lines and 64-bit capabilities.*

To allow the programmer to take advantage of the above micro-architectural changes, I expose this by introducing a new instruction, `CLoadTags` (Figure 5.2). Architecturally, it shares the format of CHERI memory instructions and returns multiple tag bits at a given address into a general purpose register. If the returned value is zero, no valid capabilities exist in this batch and the revocation routine can skip all corresponding capability words. If not, capability words with tags set are then fetched for testing. In addition, a `CLoadTags` generates tag-only memory requests to the tag controller, which means a zero return value will also avoid data traffic from DRAM to the cache hierarchy (if the corresponding data do not exist in caches).

This instruction can theoretically return XLEN number of bits, where XLEN is the width of the general purpose register. In practice, a design of this nature would complicate the implementation substantially as this many tags can span across multiple cache lines, introducing coherency problems both between the L2 and the tag cache, and between multiple cores. Eventually, I decided to restrict this instruction to return only the tags of a single cache line. First, I do not think the complexity to make this instruction coherent justifies its benefit. Second, reading tags on a larger granularity can be better achieved by another architectural assist that will be discussed next, namely the page table *cap-dirty* bits (see next section).

By restricting `CLoadTags` to a single cache line, I further implement non-temporal semantics for this instruction. `CLoadTags` is used to sweep through memory and it is not helpful to cache its response. Therefore, this instruction goes through the cache hierarchy normally but the response is not cached. This greatly reduces cache thrashing, which has significantly less performance impact for pointer-light applications.

5.4.2 Page table *cap-dirty* bit

I further take advantage of the virtual memory structure to accelerate sweeping on a page level. For almost all ISAs, virtual memory incorporates both translation and protection in the Page Table Entries (PTEs). Memory accesses are checked against multiple PTE permission bits to enforce access control, typically including read, write, execute and kernel permissions. CHERI extends the MIPS PTE to control the flow of capabilities. Two additional bits indicate whether capability loads and stores are prohibited in a page.

With this infrastructure already in place, I add a new *cap-dirty* bit. All PTEs are created with this bit cleared. The first capability store to a page will trigger a fault that is captured by the kernel, which does not terminate the program, but instead sets the bit to indicate that this page has received capability stores. With this bit set, subsequent capability writes to the page will proceed and no exception is triggered. Notice that this bit has the same behaviour as the **prohibit-cap-store** bit in the PTE entry, but is handled differently by the exception handler.

To accelerate sweeping in user space, all PTE cap-dirty bits can be written into a bit-vector structure so that a revoker can read multiple dirty bits. This complements `CLoadTags` on a much larger granularity. As each bit typically covers a 4-KiB page, a read of XLEN bits can locate (or even completely skip) potential pages of 256-KiB in memory.

However, this approach introduces the problem of false-positives. As the bit is only set on the first capability write to a clean page but not cleared after clearing all capabilities in a page, there are pages with the cap-dirty bit set but whose capabilities are no longer present. Therefore, the revoker has to ensure that whenever it discovers a false-positive page, it also clears the cap-dirty bit to reduce the false positive for subsequent revocations. Although false positives are a problem, my evaluation finds that in practice they do not appear to be a hindrance as the false positive rate is usually negligible, and a revocation pass will clear false positives anyway. This is also the reason why I only implement `CLoadTags` on a single cache line, as locating tags on a larger granularity is better achieved by this approach instead of reading tags of multiple cache lines while maintaining tag coherency.

5.4.3 Core dump study

Methodology

To allow fast prototyping and evaluation with high coverage (many applications do not yet have MIPS64 ports), I gather core dumps under x86-64 FreeBSD 12 for sweeping speed study. For the pointer distribution in data sections in a core image, I do not think that it is very different across ISAs as long as the pointer size and memory allocation strategy are the same. Therefore, I believe the core dumps obtained on x86-64 can be used to evaluate hardware revocation efficiency on CHERI-64.

I collect the core dumps by hijacking `malloc()` calls so that every nth (n depending on the revocation interval) will fork the process, and the child process aborts immediately to leave a core dump. The core dumps are then processed by analysis tools to mark pointers and calculate pointer distribution statistics. As non-CHERI architectures cannot distinguish between pointers and data, I employ heuristics to identify pointers by examining whether their values point to valid data ranges (globals, stack, heap, etc.) in the address space. This approach generally works well and is used in many conservative garbage collectors [8, 44]. I find that this algorithm usually fails to identify null pointers. However, on CHERI this is not a problem as null capabilities are of no interest to revocation.

The processed core dumps with marked pointers are transferred to the CHERI-64 FPGA. Before doing a sweep, it re-derives the marked pointers into actual capabilities to simulate its core image as if the application were compiled and run under the purecap (all pointers are capabilities) ABI. It then starts the sweeping revocation and measures its cycle and instruction count as well as cache and DRAM traffic. This allows to report the performance of the implemented architectural and micro-architectural changes on a wide range of applications.

Study of false positive pages

To deploy page table cap-dirty bits, it needs to be shown that false positives are not a major setback. A high false positive rate indicates that many pages are dirty but actually contain no capabilities, reducing its effectiveness.

To verify this, the core dumps I gathered from each application are sorted chronologically. Each core dump identifies pages which have at least one pointer. On CHERI, these pages will be marked with cap-dirty bits, and a set of cap-dirty pages is established. With n core dumps, I compare the set from core dump m with $m+1$ (m ranging from 1 to $n-1$), and the pages that are marked in set m but not in $m+1$ are false positive pages. This number is then divided by the number of total pages to derive the false positive rate.

In the evaluation, I find that the false positive rates tend to be negligible and almost zero for many applications (Table 5.1). Node.js running regexp shows a noticeable rate of 2.25%. This pattern also gives insights into how applications tend to allocate memory. Once a page is used for pointer-based data structures (graphs,

trees, linked lists, etc.), it is likely to hold similar structures in the future and not for pure data. The results indicate that in practice, false positive pages do not significantly occupy the address space, and PTE cap-dirty bits is an effective mechanism to accelerate sweeping revocation.

5.4.4 Performance of fast sweeping

I evaluated 120 core dumps from more than 50 applications, ranging from pointer-light C programs to pointer-heavy web browsers and Javascript engines. To evaluate individual hardware optimisations, I compare each with the unoptimised baseline in terms of cycle count and number of DRAM data transfers (not considering row activations), and calculate their percentage reductions. To correlate the performance with pointer distribution characteristics, three key metrics are extracted:

$$pointer\ density\ (ptr\ dens) = \frac{no.\ of\ pointers}{total\ no.\ of\ double\ words} \tag{5.1}$$

$$cache\ line\ dirty\ density\ (cache\ dens) = \frac{no.\ of\ cache\ lines\ with\ at\ least\ one\ pointer}{total\ no.\ of\ cache\ lines} \tag{5.2}$$

$$page\ dirty\ density\ (page\ dens) = \frac{no.\ of\ pages\ with\ at\ least\ one\ pointer}{total\ no.\ of\ pages} \tag{5.3}$$

Table 5.1: *Sweeping performance of sample applications. The percentages in the last four columns indicate relative* ***reduction*** *compared with the baseline. Negative means* ***overhead*** *instead of reduction. Tildes indicate negligible numbers.*

benchmark	FP rate	ptr dens	cache dens	page dens	CLoad-Tags cycle	CLoad-Tags DRAM	cap-dirty cycle	cap-dirty DRAM
sqlite3	~	0.12%	0.28%	2.56%	90.04%	98.21%	97.07%	97.41%
bsdtar-bzip2	~	1.56%	3.52%	5.81%	85.50%	95.12%	92.47%	94.75%
ssh-chacha20	~	6.73%	14.04%	20.67%	71.44%	86.41%	73.83%	83.01%
chromium-main	~	7.41%	20.62%	28.92%	64.45%	78.35%	64.90%	71.25%
chromium-renderer	0.69%	18.34%	37.15%	44.23%	44.91%	62.22%	44.79%	54.71%
js-splay	0.08%	75.15%	95.16%	96.84%	-1.35%	5.66%	1.51%	3.88%
js-early_boyer	0.62%	52.13%	66.82%	73.47%	16.96%	33.88%	16.63%	26.29%
python3	~	26.35%	63.92%	76.85%	21.41%	37.00%	17.17%	24.80%
olden-mst	~	22.03%	92.09%	93.65%	0.54%	9.19%	4.14%	7.73%
js-regexp	2.25%	24.25%	44.12%	68.46%	37.31%	55.45%	23.66%	30.98%

Several representative applications are shown in Table 5.1. Both optimisations are able to accelerate sweeping in all applications.

`CLoadTags` shows a significant performance gap between pointer-light and pointer-heavy applications. Pointer-light ones dramatically benefit from a reduction of both cycles and DRAM traffic, up to 90% and 98% respectively. `CLoadTags` allows the revoker to skip lines that contain no capabilities. Therefore, the cycle and DRAM data transfer reductions are proportional to the dirty cache line density. A minor overhead does show in *js-splay*. With almost all cache lines dirty in this benchmark, `CLoadTags` becomes pure overhead as it cannot skip but has to fetch the cache lines anyway, wasting an additional check per cache line. However, *js-splay* is a synthetic benchmark targeting object creation and destruction performance. For pointer-light and other real-world applications, this optimisation drastically improves the efficiency of sweeping revocation by taking advantage of the tag cache.

Cap-dirty bits show a similar pattern to `CLoadTags`, albeit on a larger granularity. In the evaluation, extra pages are marked dirty to represent the false positive rates, although this number is typically too low to have any effect. Also similar to `CLoadTags`, the cycle and DRAM reductions correspond well to the page dirty density.

5.4.5 Pointer concentration

Even with the same pointer density, the sweeping speed can still vary depending on other factors. One important factor is how concentrated the pointers (capabilities) are stored within cache lines due to `CLoadTags` working on a cache line granularity. Take *python3* and *olden-mst* in Table 5.1 as an example. The latter clearly has a lower pointer density but fails to show comparable sweeping speed increase with the former. Clearly, this demonstrates that even if reducing the number of pointers is not an option, developers are encouraged to concentrate the allocation of pointers to occupy fewer cache lines. This might require novel data structures and memory allocation schemes to improve over existing pointer and data layout in memory.

5.5 New allocator design for reduced sweeping frequency

5.5.1 Brief overview of dlmalloc()

The Doug Lea's Malloc (dlmalloc) [40] is widely used in systems like Android and many embedded devices, and is the basis of the allocator of the GNU C library (glibc), which in turn is commonly deployed on modern Linux systems. Two key properties of the allocator design are boundary tags and binning. The former ensures that two bordering unused chunks can be coalesced into one larger chunk, and all chunks can be traversed starting from any known chunk in either a forward or backward direction. The latter chains chunks of similar sizes and categorises them into separate bins for caching and searching. For large sizes above the `mmap_threshold`, `dlmalloc` directly manages them via `mmap` and `munmap` calls to immediately release large chunks of memory.

5.5.2 Implementation of dlmalloc_nonreuse()

Modern allocators attempt to re-allocate recently freed chunks in hope of maximising the effect of CPU caches, TLB entries, etc. However, this strategy demands extremely high numbers of sweeps to achieve a non-reuse policy. Without proper allocator support, the frequent sweeping incurs an unacceptably high cost to both run time and DRAM traffic. Therefore, I implement `dlmalloc_nonreuse` with a modified allocation policy, which avoids immediate reuse of recently freed chunks to reduce the number of sweeps required.

The original `dlmalloc` consists of two spaces: in-use chunks and free chunks in free-lists. I introduce a third quarantine space which consists of memory chunks that are freed but have not been revoked (may still have stale pointers pointing to them).

Life cycle of a memory chunk. A memory chunk enters the quarantine space when a function calls `free()`. Chunks in this state cannot be returned to a free-list unless they undergo a revocation pass. After the revocation, all stale pointers pointing to quarantined chunks are invalidated. The chunks can now be returned to a free-list and can be reused by another `malloc()` call.

Quarantine queue and threshold. The quarantine maintains a queue to include all quarantined chunks. A sweeping revocation is triggered whenever the total size of chunks in the queue reaches a threshold. The threshold is configurable at a certain percentage of the heap size. Obviously, the frequency of sweeping can be reduced at the expense of larger quarantine size.

Revocation shadow map. During sweeping revocation, stale capabilities have to be identified and revoked. To achieve this, a separate revocation shadow map is maintained to indicate whether each heap allocation granule is currently in quarantine. For each allocation granule, which I choose to be 16 bytes of memory to match the default in `dlmalloc` [40], I allocate 1 bit in a shadow map; this shadow space therefore occupies $\frac{1}{128}$ of the heap. Before a sweep, for all allocations in the quarantine buffer, the revoker "paints" the bits of the shadow map corresponding to the allocation granules to indicate that these regions are in quarantine, and references to them should be revoked in the sweep. The actual sweeping procedure performs a lookup in the shadow map using the base field of each capability to detect if it is pointing into quarantined memory[1]. If so, the capability is revoked.

Efficient shadow map lookup. To achieve high memory sweeping speeds, the shadow map lookup must be simple and efficient. By default, FreeBSD does not map the bottom 2GiB of virtual address space on 64-bit architectures. Memory mapping to the bottom 2GiB can be forced by setting the `MAP_32BIT` flag when calling `mmap()`. This default setup conveniently leaves us 2GiB for shadow maps. Therefore, all normal `mmap`s and `munmap`s in the allocator are accompanied by a shadow space `mmap` call with `MAP_32BIT` set under 2GiB. Since the heap allocation granule is 16 bytes, a shadow map accompanying a normal `mmap` is only $\frac{1}{128}$ in size. Also, I set the `MAP_FIXED` flag so that the shadow map has to be at a constant bit shift (a right shift of 7 for 16-byte granule) of the original. This shadow map scheme allows fast, flat index lookup for testing each capability reference during a sweep, and is deterministic in its instruction count.

The following shows the C code for revoking stale capabilities within a region. Note that in a CHERI system, `uintptr_t` type is a capability when the tag is set.

[1]We can be sure that any heap capability will have a base within the original `malloc` bounds due to the monotonicity of capabilities.

```
1 for(uintptr_t* x=MIN_ADDR; x<MAX_ADDR; x++) {
2     uintptr_t capword = *x;
3     if(is_capability(x)) {
4         capword >>= 4; // 16-byte alloc granule
5         // Get the bit index.
6         int bitIdx = capword & 0x7;
7         // Get the byte from the shadow space at a constant
             ↪shift.
8         char shadowbyte = *(char*)(capword >> 3);
9         if(shadowbyte & (1<<bitIdx)) {
10             // Pointing at freed memory.
11             // Invalidate the capability.
12             *x = 0;
13         }
14     }
15 }
```

Parallel sweeping. Unlike tracing garbage collection which requires a tree walk from the roots to find reachable objects which exposes only limited amount of parallelism, sweeping on the other hand is embarrassingly parallel. As shown in the code snippet of shadow map lookup, there are no dependencies between any two iterations of the for loop, thus theoretically all iterations can be performed in parallel. As a result, many optimisations are possible to achieve a higher (or completely saturate) DRAM bandwidth of the system during sweeping. For example, programs on a multi-core machine can spawn multiple sweeper threads to sweep its address space in parallel; dedicated DMA engines can perform sweeping, shadow map lookup and capability invalidation in the background. Even in this chapter where the benchmarks are mostly single-threaded and I assume only a single core model, the operations of sweeping can fit in vector instructions which is able to parallelise nicely even on a single core.

Coalescing chunks in the quarantine space. Each quarantined chunk has an extra function call overhead, as each `free()` first calls `quarantine()` before the revoker calls the actual `free` function on each chunk after revocation. In my implementation, calling `quarantine()` is significantly cheaper than the actual freeing because the only maintenance is the quarantine queue, whereas the actual `free` needs to find the correct bucket (which may involve a prefix tree walk) or to perform other maintenance work. Therefore, if quarantined chunks are coalesced before calling `free`,

the number of eventual calls is lower than calling `free` directly, which results in higher performance.

The coalescing algorithm is similar to how `dlmalloc` coalesces free chunks. The 16-byte alignment of memory chunks leaves 4 bits for other purposes in the size field of chunk headers, among which 2 are already in use by `dlmalloc`. I use the remaining 2 bits to indicate whether this chunk and the previous chunk is in the quarantine space (**cdirty** and **pdirty** respectively). In this way, it can be trivially determined whether adjacent chunks are also in the quarantine space, in $O(1)$ complexity.

```
1   void free_nonreuse(void* mem) {
2       void* ptr = mem2chunk(mem); // Point to metadata.
3       size = ptr->size;
4       if(ptr->pdirty) { // Try to coalesce with the previous
            ↪chunk.
5           pptr = get_prev_chunk_ptr(ptr);
6           unlink_freebuf(pptr);
7           size += pptr->size;
8           ptr = pptr;
9       }
10      nptr = get_next_chunk_ptr(ptr);
11      if(nptr->cdirty) { // Try to coalesce with the next chunk.
12          unlink_freebuf(nptr);
13          size += nptr->size;
14      }
15      // Write metadata and insert to quarantine.
16      ptr->size = size;
17      ptr->cdirty = 1;
18      get_next_chunk_ptr(ptr)->pdirty = 1;
19      insert_freebuf(ptr);
20  }
```

As described in the C code, adjacent quarantined chunks are removed from the queue, and a new coalesced chunk will be formed and added to the tail of the queue (also demonstrated in Figure 5.3).

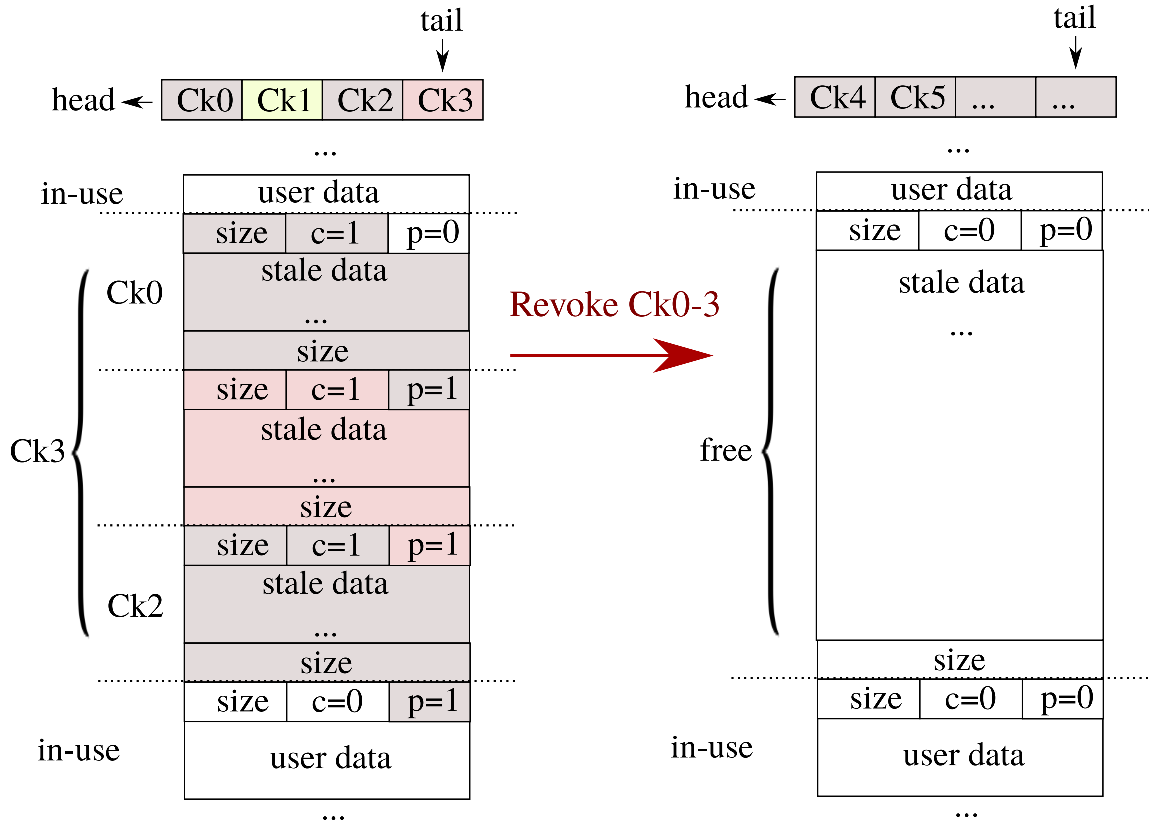


Figure 5.3: ***c*** *and* ***p*** *represent* ***cdirty*** *and* ***pdrity*** *bits. At the end Ck0 and Ck2 are unlinked from the queue and the newly freed chunk (in the middle) will be coalesced with Ck0 and Ck2 into Ck3 and inserted at the tail of the queue. After a revocation, Ck3 is returned to the free lists to be reused.*

5.5.3 Experimental setup

In addition to evaluation of the hardware extensions on the CHERI FPGA platform in the previous core dump study, I have designed experiments of the new allocator on a modern x86-64 machine to establish performance expectations for a wide deployment of mature CHERI implementations. Memory-sweeping performance depends heavily on the microarchitecture. These experiments allow us to characterise revocation using state-of-the-art memory systems, vector extensions, and out-of-order superscalar hardware. The x86-64 platform also has much higher application and benchmark coverage.

The x86 machine has the following specifications: Intel Core i7-7820HK CPU, 2.9GHz, 4 cores 8 threads, 8MiB LLC, 14–18 stage out-of-order superscalar pipeline, AVX2 support, 16GiB DDR4 2400, FreeBSD 12.0.

Allocator override. All applications and benchmarks need to be run with `dlmalloc_nonreuse()` instead of the default libc memory allocator (`jemalloc` on FreeBSD 12.0). Fortunately, this can be done easily by compiling the allocator into a shared library and overriding with `LD_PRELOAD`. In addition, I implement a debug mode to collect allocation traces to analyse free rate, sweep rate, heap layout, etc.

Sweeping cost. `dlmalloc_nonreuse()` evaluates all overheads besides the sweeping itself. This is because on a non-capability architecture, sweeping is impossible without the ability to precisely identify pointers. Instead, I again collect memory core dumps in a similar way to the evaluation of hardware optimisations at sweeping points. I preprocess the memory image so that non-pointers are zeroed, thus a `CGetTag` can be simulated by a comparison to zero. The core dump also preserves the revocation shadow map, which is used during the sweep. The sweeping is then done offline on the preprocessed core dumps. It simulates a system API that returns an array of pages that could contain capabilities according to PTE cap-dirty flags. While page elimination can be modelled sufficiently on a standard microarchitecture, `CLoadTags` is not modelled due to the lack of tagged memory and a tag cache. As a result, the performance numbers presented later are a pessimistic estimation of the full optimisations possible on CHERI. To evaluate the overall cost, I perform revocation sweeps on ten sample core dumps from across each application's execution. I then multiply the average sweep time by the total number of sweep events to derive the total sweeping cost for that execution.

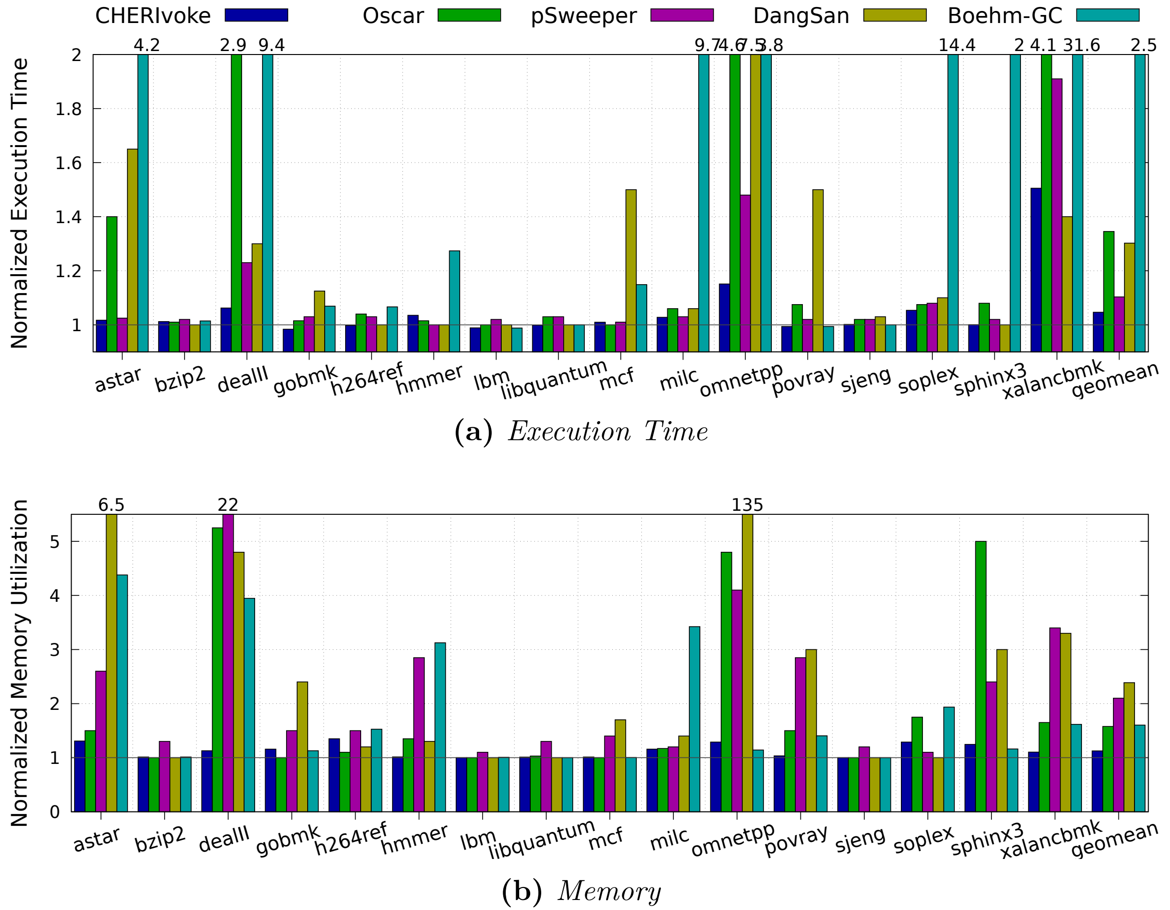
Vectored sweeping. As sweeping is embarrassingly parallel, parallelisation opportunities exist even on a single core using vector instructions. Most modern CPUs of different ISAs have decent vector extensions, and it is reasonable to imagine that a commercial CPU with CHERI support is able to operate on capability vectors. Therefore, to achieve higher sweeping performance on the x86 machine, I convert the naïve sweeping loop into vectored code in AVX2 instructions.

```
1 static inline void
2 sweep_page(char* thisPage) {
```

```
3   for(__m256i* ptr = (__m256i*)thisPage; (char*)ptr<thisPage
      ↪+4096; ptr++) {
4     __m256i zeroVec = _mm256_setzero_si256();
5     // Use streaming instructions for less cache disturbance.
6     __m256i loadVec= _mm256_stream_load_si256(ptr);
7     // a mask indicating which are capabilities
8     __m256i ptrMask= _mm256_cmpgt_epi64(loadVec, zeroVec);
9     // Heap granularity is 16 bytes, shift by 4.
10    loadVec = _mm256_srli_epi64(loadVec, 4);
11    // A mask to select the bot 6 bits.
12    __m256i botMask = _mm256_set1_epi64x((size_t)0x3f);
13    __m256i bitShift = _mm256_and_si256(loadVec, botMask);
14    // Now pointing to 64-bit aligned addresses in shadow space.
15    loadVec = _mm256_srli_epi64(loadVec, 6);
16    loadVec = _mm256_slli_epi64(loadVec, 3);
17    // Do a masked gather.
18    __m256i shadowBits = _mm256_mask_i64gather_epi64(zeroVec,
        ↪NULL, loadVec, ptrMask, 1);
19    shadowBits = _mm256_srlv_epi64(shadowBits, bitShift);
20    __m256i ones = _mm256_set1_epi64x((size_t)0x1);
21    shadowBits = _mm256_and_si256(shadowBits, ones);
22    shadowBits = _mm256_slli_epi64(shadowBits, 63);
23    // Zero stale capabilities.
24    _mm256_maskstore_epi64(ptr, shadowBits, zeroVec);
25  }
26 }
```

Notice that the entire sweeping procedure can be expressed purely in vector instructions (except for the outer for loop, of course). This demonstrates how sweeping has significant parallelism potential in a way that garbage collection cannot match. With multiple cores and DMA engines in practice, sweeping should be limited only to the DRAM bandwidth of the system.

Benchmarks. I evaluated benchmarks taken mostly from SPEC CPU2006 [33], in line with other papers. The subset includes all SPEC-CPU2006 benchmarks that would compile under x86_64 FreeBSD, which is the infrastructure for CHERI research: astar, bzip2, gobmk, h264ref, hmmer, lbm, libquantum, mcf, milc, povray, sjeng, soplex, and sphinx3. In each case, I evaluated on the SPEC reference input. The benchmark set further adds ffmpeg, which has a larger allocation throughput than



(a) *Execution Time*

(b) *Memory*

Figure 5.4: *Overheads compared with results reported by other state-of-the-art techniques. CHERIvoke represents the new dlmalloc.*

any SPEC benchmark and is useful to more fully account for worst-case application behavior. I take the average of 4 runs for each benchmark.

5.5.4 Overall overheads of dlmalloc_nonreuse()

The overall observed overhead is shown in 5.4, compared with other literature [8, 19, 43, 38] that do not make use of CHERI capabilities. I am very grateful to Sam Ainsworth for providing the results and insights of other non-CHERI temporal-safety schemes used in comparison. For a target 25% heap storage overhead in the quarantine buffer, sweeping revocation adds 4.7% execution time and 12.5% total memory overhead on average. This significantly outperforms any other technique. Further, capability revocation performs far more reliably, with only 1.51× and 1.35× maximum

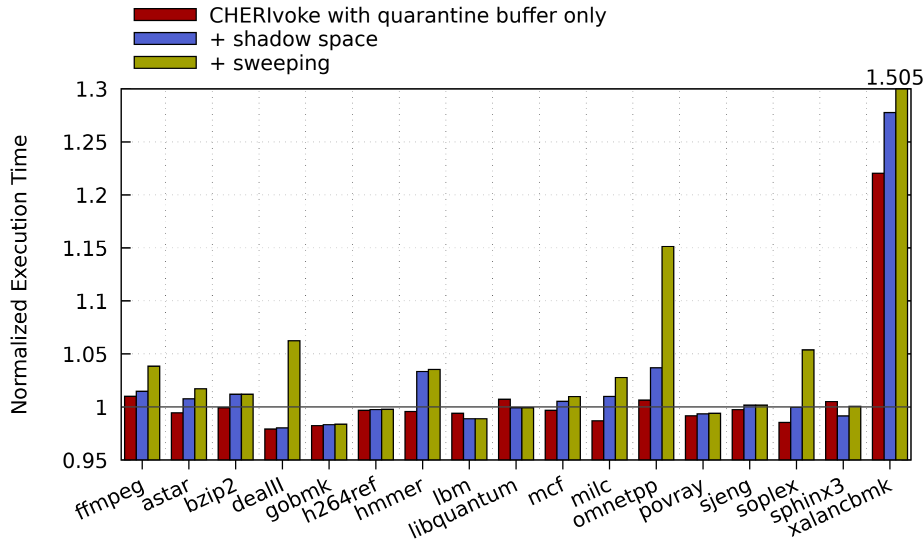


Figure 5.5: *Run-time overhead decomposition for the constituent parts, with the default 25% heap overhead.*

runtime and memory overheads. Sweeping revocation with `dlmalloc_nonreuse()` has significantly more predictable behaviour regardless of workload, as its sweeping technique suffers none of the worst cases encountered by more complex temporal-safety schemes: overheads are proportional to memory freed and pointer density, rather than pointer movement, number of frees, loads per second, or memory layout.

5.5.5 Breakdown of overheads

Figure 5.5 shows overheads for successively adding constituent parts of sweeping revocation, beginning with quarantining freed memory, adding shadow-map maintenance, and finally, full-memory sweeps. While memory sweeping is usually the dominant overhead, there are notable exceptions that are discussed below.

Quarantine buffer. `dlmalloc` aggressively reuses recently freed memory for better cache efficiency. Introducing a quarantine buffer misses the opportunity to reuse cached memory. The quarantine buffer has negligible impact on most benchmarks. For xalancbmk, however, the quarantine buffer increases execution time by 22%. Performance counters confirm that instruction count only grows by 3%, but L2 cache misses grow by 50%.

The quarantine buffer actually improves performance in most of the benchmarks. One reason for this is the coalescing strategy implemented in Section 5.5.2. The dealII benchmark, for example, has 630,000 calls to free per second, constituting a significant amount of execution time. `dlmalloc_nonreuse()` simply quarantines these allocations at typically less than half the execution time of a real free. If these freed regions coalesce well, many fewer free operations will be performed when the quarantine buffer is drained than would have been performed on demand. While this effect is minor, only a few benchmarks that gain advantage from the addition of the quarantine buffer subsequently experience a net overhead when considering shadow-map maintenance and full memory sweeps.

Shadow-map maintenance. Revocation also requires maintenance of the revocation shadow map (the second bar in Figure 5.5). The size of the shadow map is small compared to the heap itself. Therefore, the net impact of shadow-space maintenance is minor for all applications benchmarked.

Overhead of sweeping. The overhead of sweeping itself can be easily observed from the difference between the second and third bar in Figure 5.5. For allocation intensive benchmarks, the largest cost is in memory sweeping. It is shown that, of the four benchmarks that have overheads beyond 5%, omnetpp, dealII and soplex are dominated by sweeping overhead and xalancbmk is a special case, as discussed above.

Optimisation of sweeping. The majority of the overhead comes from sweeping; critical to that is how fast we are able to move through memory. In Figure 5.6, I evaluate the performance of three different sweeping-procedure kernels, implemented under increasing levels of complexity. A naïve sweeping loop (presented in red) comes far short of saturating the full system DRAM bandwidth of 19,405MiB/s (saturating only 28% of the full bandwidth on average). I optimise the naïve loop by unrolling and manually pipelining for the target micro-architecture to achieve better instruction scheduling (32% average). Further, I evaluate the AVX2 vectorised sweeping procedure presented in Section 5.5.3, which shows the highest bandwidths (39% average).

The naïve loop (see code in Section 5.5.2) compiles into 15 instructions, among which three are conditional jumps. The lack of instruction-level parallelism within each loop and the unpredictability of the first two jumps (conditioning on whether a capability word is valid and whether the shadow space bit is set) result in a poor Instruction Per Cycle (IPC). The manual optimisation unrolls two loops at once,

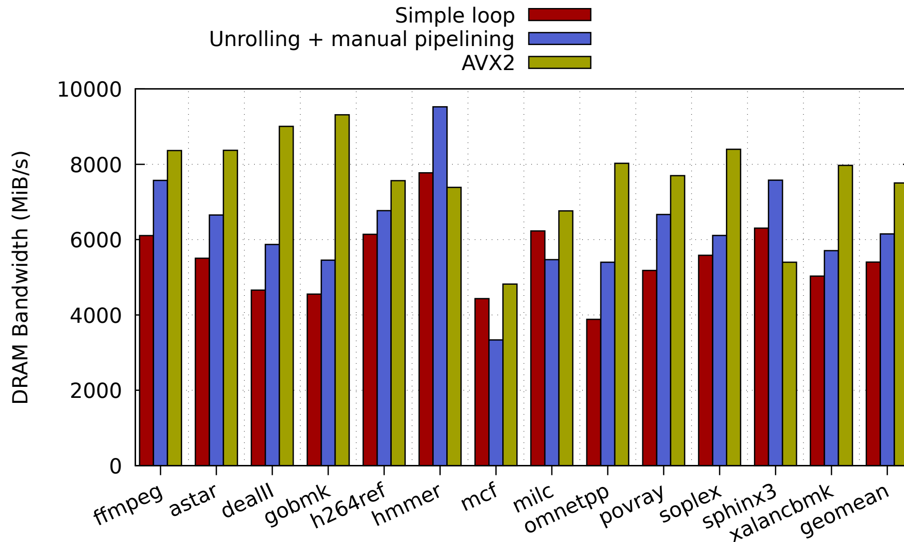


Figure 5.6: *Memory bandwidth achieved for the sweep loop with different optimisations. The system's full bandwidth is 19405MiB/s.*

interleaving the instructions (especially between dependent instructions of each loop) to increase pipeline utilisation and to hide memory latency. Of course, modern out-of-order CPUs like the one in this evaluation can already extract some ILP across loops even in naïve sweeping, so manual optimisation is only mildly effective.

AVX2 significantly changes the sweeping loop. Being able to fully parallelise using vector instructions, the sweeping procedure can process an entire cache line in only 29 instructions (see code in Section 5.5.3). In contrast, one cache line takes around 120 and 116 instructions for the naïve loop and manual optimisation respectively. Further, all unpredictable branches are replaced with conditional vector instructions which avoid mis-speculations. However, several factors limit how AVX2 sweeping performs [55]:

- Intel AVX instructions typically run at throttled frequencies to prevent thermal damage, whereas non-AVX instructions benefit from frequency boosts.
- The CPU operates in frequency phases, which resumes at normal frequencies after a delay when transitioning from AVX to scalar operations.
- AVX instructions typically exhibit higher latencies. For example, a gather operation has a latency of ~20 cycles and an IPC of less than 0.25.

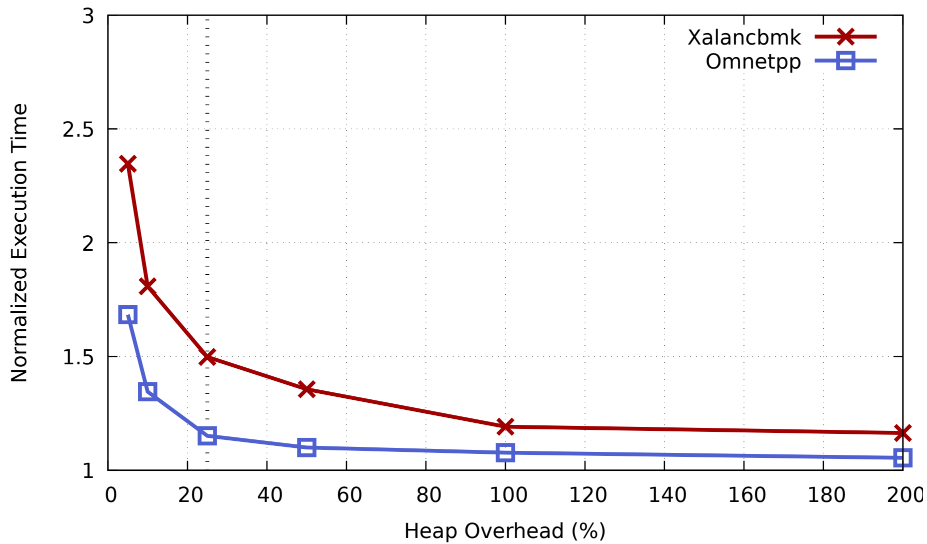


Figure 5.7: *Normalised execution time for the two workloads with highest overheads, at varying heap overhead. Default setup shown by dotted line.*

These restrictions on AVX instructions as well as the limit on the memory subsystem mean that ultimately, the sweeping speed increase is around 40% faster than the naïve baseline or 22% faster than the optimised sweeping loop as the evaluation shows, not as dramatic as the instruction count reduction may suggest.

5.5.6 Tradeoff between space and time

Another advantage of sweeping revocation is that time and space overheads can be traded off for one another. Under the default quarantine size of 25% of the heap, xalancbmk and omnetpp show the highest overheads among all benchmarks. I re-evaluate both with different target heap-space overheads. Figure 5.7 illustrates the results. The higher the heap overhead, the less of a performance impact it will observe, even on highly allocation-intensive workloads.

There are two reasons for this. Clearly, a larger quarantine will be filled less frequently than a smaller one, resulting in fewer sweeps. This accounts for the majority of the performance increase, as most of the overhead is brought about via the sweeping procedure. The second is more subtle: for xalancbmk, by the time it reaches 100% heap overhead, the normalised execution time is actually lower than the non-sweeping

costs alone in Figure 5.5. A consistent reduction in non-sweeping overheads is found as the buffer increases, corresponding to an increase in observed cache hit rate for the program. This counter-intuitive result is caused by better allocation-fragmentation properties as the heap size is increased: under severe temporal fragmentation, it is better to quarantine memory for longer to allow cache lines to fall entirely out of use rather than frequently releasing small fragments in a severely fragmented heap.

5.6 Alternative sweeping schemes and concurrency

This section briefly introduces discussions and proposals towards other schemes of sweeping revocation and concurrent revocation. Many people working on temporal safety, including Nathaniel Filardo, Jonathan Woodruff, Lucian Paul-Trifu, Peter Rugg, Hadrien Barral, Robert N. M. Watson and myself, have contributed to the ideas and implementation.

5.6.1 Subset testing revocation

An alternative approach prior to shadow map revocation is to simply sweep through memory and test whether each memory word is a subset to the memory region the allocator is revoking. If so, the capability is stale and therefore invalidated. To accelerate, a new instruction `CTestSubset` is introduced to atomically test whether a capability is a subset of the other.

Advantages. This model is relatively simple without any shadow space maintenance and "painting". The only overhead is maintaining the quarantine buffer.

Shortcomings. With a certain quarantine buffer size, the frequency of sweeping in this scheme depends on the number of quarantined chunks rather than the quarantine size itself, which exhibits terrible performance in worst cases. Each sweep dequeues only one (or a small fixed number) chunk in the quarantine instead of returning all of them in the shadow map approach. Another problem of this appoach is the need to reduce temporal fragmentation to maximise quarantine buffer coalescing. If the quarantined chunks are adjacent and can be coalesced well, the number of chunks in the quarantine can be made small, thus reducing the sweeping frequency. However, in earlier studies we find that this is not always true, and we often need to

apply heuristics to place allocations with similar lifetimes together, in hope that they will be freed at similar times in the future for maximised coalescing. This results in a complex design and unpredictable performance. However, for embedded applications where dynamic allocations are predictable and not often, subset testing revocation might already be sufficient.

5.6.2 Concurrent revocation

The allocator so far assumes a stop-the-world model, i.e., no mutators can run concurrently when the revoker is working. Similar to other language virtual machines and runtimes, a revocation pass may pause the program for an undeterministic period of time, which is unacceptable for many low-latency use cases.

Similar to concurrent garbage collection, one may relax the rules and still permit stale capabilities to be stored. At the beginning, the revoker clears all cap-dirty bits in page table entries. At the end of a pass, the revoker scans all cap-dirty bits to know which pages have new capability stores. The set of new dirty pages should be small enough to allow a quick stop-the-world pass. Also, invalidating each capability requires LL/SC sequences. If the revoker decides a capability is invalid, a mutator may in parallel rewrite it with valid data, and the revoker should not invalidate the new data.

5.7 Summary

This chapter described my work on CHERI temporal memory safety. To improve the efficiency of sweeping revocation, I focused on two ways to reduce the overhead. One is to accelerate memory sweeping by introducing architectural changes and micro-architectural optimisations. The other is to implement a non-reuse memory allocator that uses quarantine buffers and shadow maps for reduced sweeping frequency. Together, they make it feasible to deploy CHERI temporal safety in most applications, giving much lower and more predictable performance overhead compared with other state-of-the-art probabilistic temporal defenses.

Chapter 6

Conclusion

6.1 Contributions

In this dissertation, I reviewed the state-of-the-art memory safety for embedded processors, drawing the first conclusion that existing security approaches are insufficient. They suffer from high run-time overhead, high latency and non-determinism and fail to scale as embedded devices become more capable and ubiquitous. Unlike conventional embedded CPUs, the 64-bit CHERI ISA provides direct hardware support for fine-grained memory protection, scalable compartmentalisation and fast domain crossing. I hypothesised that CHERI would provide a novel approach to solving the existing memory safety problems that could be applied to 32-bit embedded processors, as well as offering significantly improved scalability compared with the state-of-the-art. I addressed two major problems in applying CHERI capabilities to embedded processors. Firstly, typical 32-bit embedded processors do not function with existing capability compression schemes for 64-bit CPUs. I implemented and evaluated a new 64-bit compressed capability scheme tailored to embedded use cases. Secondly, the OS used in previous CHERI research, CheriBSD, was a heavy-weight UNIX system with POSIX abstractions and virtual memory, which was obviously not suitable for embedded devices. As a result, I designed and implemented a real-time kernel specifically for resource-constrained, latency-sensitive processors in a flat physical address space (no MMU), using capabilities as the only mechanism to enforce fine-grained memory protection and scalable task isolation.

I further hypothesised that the monotonicity and unforgeability properties of capabilities indirectly enable temporal memory safety, deterministically solving problems like use-after-free vulnerabilities. However, frequently sweeping memory to invalidate capabilities could incur infeasible overhead. To address this, I introduced new instructions, hardware optimisations and new memory allocator designs to significantly reduce the cycle overhead and DRAM traffic for capability sweeping revocation, making it practical for most applications to adopt CHERI temporal memory safety.

6.1.1 A capability format for 32-bit cores

For 64-bit CPUs, the original uncompressed CHERI-256 quadrupled the size for all pointers which was unacceptable. We developed the CHERI Concentrate format to compress pointer, bounds, permissions and so forth within 128 bits, which was achieved with reasonable pipelining complexity while maintaining legacy C compatibility. For typical 32-bit embedded processors, I further compressed the metadata and built CHERI-64 to make CHERI capabilities feasible for 32-bit processors. My study showed that memory fragmentation was still minimum under heavy compression. The CHERI-64 implementation replaced the conventional MPU/PMP unit with the capability coprocessor. To evaluate the hardware costs, I compared CHERI-64 hardware logic utilisation against the RISC-V PMP, concluding that it was at a comparable cost with state-of-the-art hardware security components in embedded processors. The results also showed that CHERI-64 had much less impact on the critical path, whereas the large number of associative comparisons in MPU and PMP models incurred significant penalties on pipelining and core clock frequency.

6.1.2 CheriRTOS

Real-time operating systems atop conventional embedded architectures heavily rely on MPU/PMP models and TrustZone to enforce isolation and memory safety. As described in Section 2.3.2 and 2.4, those protection schemes unfortunately suffer from several drawbacks including the coarse granularity, high latency and poor scalability. In contrast, CHERI primitives can resolve the isolation and memory protection problems efficiently and scalably. Therefore, I designed and implemented a CHERI-based real-time operating system, CheriRTOS, incorporating CHERI capabilities from the

ground up for scalable task isolation, fine-grained memory protection and secure inter-task communication. The evaluation confirmed that these benefits could be achieved without violating the low-latency and determinism constraints. In fact, unlike previous memory safety measures that brought complexity and overhead, many aspects of a CHERI-based kernel actually offered improvements in performance and design. For instance, the offset addressing in PCC and DDC rendered Position Independent Code (PIC) unnecessary, which increased performance; the scalable task isolation and low-latency domain crossing facilitated a decentralised kernel where drivers from various vendors could be efficiently isolated in constrained compartments; a fine-grained heap brought scalability and relieved programmers of the tedious and costly manual bounds checks. These benefits were made possible with a CHERI-based kernel and CHERI primitives.

6.1.3 CHERI temporal memory safety

As previously described, CHERI can theoretically guarantee temporal safety by frequently sweeping memory and invalidating stale capabilities. Due to the inefficiency of the naïve approach, I investigated the means to optimise the key factors in the overhead of sweeping revocation. I proposed architectural and micro-architectural modifications that substantially improved the efficiency of memory sweeping, especially taking advantage of the tag cache and page table structure to avoid unnecessary memory traffic. The proposed changes were implemented on the FPGA which showed that the processor cycles and DRAM traffic overhead due to sweeping could be reduced by orders of magnitude, especially for pointer-light applications. On the software side, I implemented an alternative memory allocator which used quarantine buffers to buffer and coalesce freed memory regions before revocation. I explored the tradeoffs between quarantine buffer overhead and sweeping frequency. The hardware and software assists already made it feasible for most applications to adopt temporal memory safety under CHERI, and paved the way towards incremental and concurrent capability revocation in the future as well.

6.2 Future work

6.2.1 CHERI-64

To maintain toolchain compatibility, the permission bits of CHERI-64 are not well compressed. In a simpler model, the possible use cases and combinations of permissions should only be a subset of what we have seen in CheriBSD. Clearly, the 1024 combinations of 10 permission bits in CHERI-64 are likely to be much higher than needed in practice. With careful profiling and design, more bits could be extracted to be used to increase precision of bounds or the size of the object type field.

A split register file with separate capability registers adds latency to context switches. Merging the coprocessor with the main pipeline and extending existing registers to also hold capabilities can further reduce logic and latency overhead. This has already been worked on in our new CHERI-RISC-V implementation.

6.2.2 CheriRTOS

The traditional MIPS pipeline cannot handle complex register loads and stores in a single cycle, thus the trusted stack helper is implemented purely in software. If the `CCallFast` instruction was used in a frequent and fine-grained way, this overhead might be worth optimising. We have seen ARM Cortex-M embedded processors performing hardware-assisted register stacking and restoring on context boundaries. If hardware-assisted trusted stacks are implemented, the latency of `CCallFast` in CheriRTOS is expected to be much closer to the baseline.

6.2.3 CHERI temporal memory safety

I provide temporal memory safety through revocation of stale capabilities using a stop-the-world revoker. While more complex, it may be preferable to undertake revocation concurrently with mutator threads as used by concurrent garbage collectors. The question is how to guarantee correctness while still permitting the mutators to proceed. It is expected that a mature concurrent revocation scheme requires more

hardware assists and software changes (especially kernel/OS support) than what we have seen at this point.

6.2.4 Adopting capability protection in future embedded devices

This dissertation has explored the design space and has demonstrated an implementation that facilitates efficient, scalable and fine-grained memory protection for embedded processors using CHERI capability protection. The security guarantees and the low overhead are able to justify its deployment in future devices. However, similar to other emerging architectures, CHERI requires the following steps before achieving wider adoption in commercial and industrial embedded systems.

Firstly, researchers and engineers must embrace CHERI as a new paradigm. Often, security measures are introduced incrementally as extensions to ease the transition and deployment in industry. However, CHERI fundamentally changes several key aspects which create new ideas and concepts that cannot be viewed as incremental to conventional architectures. As a result, a wider understanding of CHERI is a prerequisite. For example, the idea of sealed capabilities facilitating intra-address-space domain transitions can be quite obscure to a non-CHERI audience. To educate people about CHERI capabilities, we need to engage them more frequently through technical talks, conferences, laboratory sessions and industrial collaborations to ensure that CHERI is well understood.

Secondly, unlike other security components that function as add-ons like MPU and Trustzone, CHERI interacts intricately with the main CPU, memory hierarchy as well as the operating system. The design and optimisation of CHERI in embedded processors cannot be standalone to the rest of the system. Although the evaluation suggests the total transistor logic and run-time overhead should be reasonable, we can imagine the effort it takes to integrate CHERI into a complete system. The higher difficulty in integration translates to a longer wait-time before wider adoption.

Thirdly, the maturity and availability of CHERI platforms must be improved. Currently, the CHERI ecosystem is still maturing with ongoing work in the ISA specification, compiler toolchain, operating system, formal verification and so forth. Also, official CHERI platforms are not readily available to the public for experimentation and development. What we can learn from the successful ARM embedded ecosystem

is that a mature toolchain and hardware platform must be ready before attracting developers and wider adoption.

6.2.5 Extrapolating to non-CHERI systems

Although this research work bases itself on CHERI capabilities, the results and conclusions are not necessarily restricted to the CHERI world. The findings can be easily extrapolated to other architectures and security related topics. Typical examples are listed below.

CHERI-CC compression algorithm. The compression algorithm should be applicable to all object-bounds related metadata. MPU regions are a perfect candidate to apply compression due to its large number of bounds registers. Compressing multiple registers of a region into one significantly reduces the context size, which potentially allows for per-task MPU context. Similarly, Intel MPX observes performance degradation due to the contention on bounds registers. The compression algorithm will greatly alleviate the pressure on the bounds register file, translating to better performance.

Temporal memory safety wrt. spatial memory protection. Chapter 5 revealed how temporal memory safety should be built atop a spatial memory safety infrastructure. Although drawn on CHERI, the conclusion of the relationship between the two aspects of memory safety is applicable to other schemes. With bounds and monotonicity, most other spatial schemes can adopt similar architectural optimisations and memory allocator design to construct a low-overhead defense against temporal vulnerabilities. Of course, CHERI shines in its unforgeability, which means the temporal safety built on top it is deterministic, unlike other probabilistic defenses.

6.2.6 Adversarial security evaluation

The spatial and temporal safety guarantees presented in this thesis demonstrably improve resistance to many classes of attacks. Once these attacks have been rendered ineffective, there remains the question: what new attacks will be invented to circumvent the new protection mechanisms? The adversarial analysis is left as future work.

Appendix A

CHERI Concentrate bounds and region arithmetic

This appendix assumes a 32-bit address space and a capability precision of 8. Lower case and upper case letters represent the actual values and the encoded fields in the capability format respectively. For example, a capability has a base b of `0x1200`, but due to precision, only 8 bits can be encoded in the B field as `0x12`.

T/t denotes top. B/b denotes base. A/a denotes the pointer field. L/l denotes length. E/e denotes exponent. R/r denotes the representable limit.

A.1 Encoding the bounds

CHERI ISA adds the `CSetBounds` instruction to allow selecting the appropriate precision for a capability. It takes the full pointer address a as the desired base, and takes a length operand from a general-purpose register, thus providing full visibility of the precise base and top to a single instruction – which can select the new precision without violating a tenet of MIPS (our base ISA) by requiring a third operand.

Deriving E

The value of E is a function of the requested length, l:

$$\text{index_of_msb}(x) = \text{size_of}(x) - \text{count_leading_zeros}(x)$$
$$E = \text{index_of_msb}(l[31:8])$$

This operation chooses a value for E that ensures that the most significant bit of l will be implied correctly. If l is larger than 2^8, the most significant bit of l will always align with $T[7]$, and indeed $T[7]$ can be implied by E. If l is smaller than 2^8, E is 0, giving more bits to T and B and so enabling proportionally more out-of-bounds pointers than otherwise allowed for small objects.

We may respond to a request for unrepresentable precision by extending the bounds slightly to the next representable bound, or by throwing an exception. These two behaviors are implemented in the `CSetBounds` and `CSetBoundsExact` variants respectively.

Extracting T and B

The `CSetBounds` instruction derives the values of B and T by simply extracting bits at E from b and t respectively (with appropriate rounding):

$E = 0$	$E > 0$; $T[1:0]$ and $B[1:0]$ implied 0s
$T = t[6:0]$	$T[6:2] = t[E+6:E+2] + \text{round}$
	$\text{round} = \text{one_if_nonzero}(t[E+1:0])$
$B = b[8:0]$	$B[8:2] = b[E+8:E+2]$

Rounding Up *length*

The `CSetBounds` instruction may round up the top or round down the base to the nearest representable alignment boundary, effectively increasing the length and potentially increasing the MSB of length by one, thus requiring that E increase to ensure that the MSB of the new L can be correctly implied. Rather than detect whether overflow will certainly occur (which did not pass timing in our 100MHz CHERI-128 FPGA prototype), we choose to detect whether $L[7:3]$ is all 1s – i.e., the

largest length that would use this exponent – and force T to round up and increase E by one. This simplifies the implementation at the expense of precision for 1/16th of the requestable length values.

A.2 Decoding the bounds

Unlike Low-fat, CHERI Concentrate can decode the full t and b bounds from the B and T fields even when the pointer address a is not between the bounds. We now detail how each bit of the bounds is produced:

Lower bits: The bits below E in t and b are zero, that is, both bounds are aligned at E.

Middle bits: The middle bits of the bounds, $t[E+8:E]$ and $b[E+8:E]$, are simply T and B respectively, with the top two bits of T reconstituted as in Chapter 3. In addition, if I_{E} is set, indicating that E is stored in the lower bits of T and B, the lower two bits of T and B are also zero.

Upper bits: The bits above $E+8$, for example $t[31:E+9]$, are either identical to $a[31:E+9]$, or need a correction of ± 1, depending on whether a is in the same alignment boundary as t, as described below and in Chapter 3.

Deriving the representable limit, R

CC allows pointer addresses within a power-of-two-sized space, $space_{\mathrm{R}}$, without losing the ability to decode the original bounds. The size of $space_{\mathrm{R}}$ is $s = 2^{E+9}$, fully utilizing the encoding space of B. Figure A.1 shows an example of object bounds within the larger $space_{\mathrm{R}}$. Due to the extra bit in B, $space_{\mathrm{R}}$ is twice the maximum object size (2^{E+8}), ensuring that the out-of-bounds representable buffers are, in total, at least as large the object itself.

As portrayed in Figure A.1, $space_{\mathrm{R}}$ is not usually naturally aligned, but straddles an alignment boundary. Nevertheless, as $space_{\mathrm{R}}$ is power-of-two-sized, a bit slice from its base address $r_b[E+9:E]$ will yield the same value as a bit slice from the first address above the top, $r_t[E+9:E]$. We call this value the *representable limit*, R.

Locating b, t, and a either above or below the alignment boundary in $space_R$ requires comparison with this value R. We may choose R to be any out-of-bounds value in $space_R$, but to reduce comparison logic we have chosen:

$$R = \{B[8:6] - 1, zeros\text{'}6\}$$

This choice ensures that R is at least 1/8 and less than 1/4 of the representable space below b, leaving at least as much representable buffer above t as below b.

For every valid capability, the address a as well as the bounds b and t lie within $space_R$. However the upper bits of any of these addresses may differ by at most 1 by virtue of lying in the upper or lower segments of $space_R$. For example, if a is in the upper segment of $space_R$, the upper bits of a bound will be one less than the upper bits of a if the bound lies in the lower segment. We can determine whether a falls into upper or lower segment of $space_R$ by inspecting:

$$A_{\text{mid}} = a[E+8:E]$$

If A_{mid} is less than R, then a must lie in the upper segment of $space_R$, and otherwise in the lower segment. The same comparison for T and B locates each bound uniquely in the upper or the lower segment. These locations directly imply the correction bits c_t and c_b that are needed to compute the upper bits of t and b from the upper bits of a.

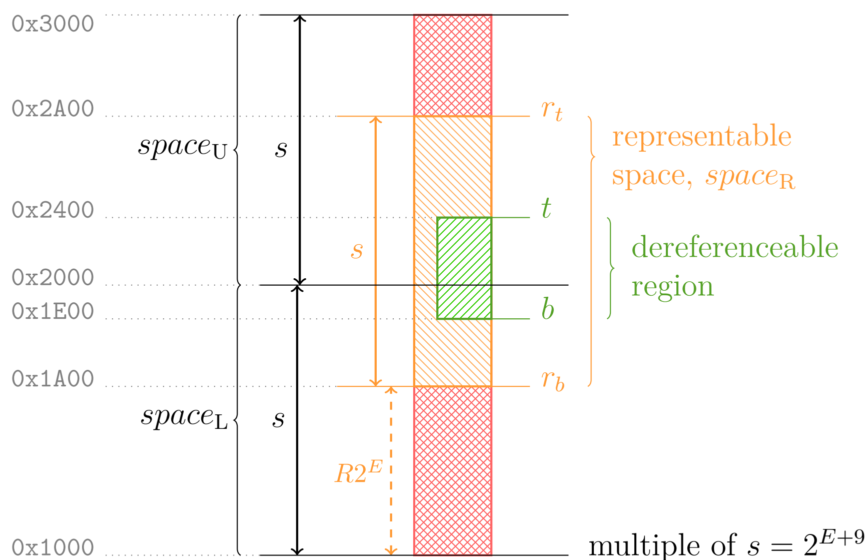


Figure A.1: *CHERI Concentrate bounds in an address space. Addresses increase upwards. To the left are example values for a `0x600`-byte object based at `0x1E00`.*

As we have chosen to align R such that $R[5:0]$ are zero, only three-bit arithmetic is required for this comparison, specifically:

$$a_in_upper_segment = A_{\text{mid}}[8:6] < R[8:6]$$

While Low-fat requires a 6-bit comparison to establish the relationship between a, t, and b, growing with the precision of the bounds fields, CC requires a fixed 3-bit comparison regardless of field size, particularly benefiting CHERI-128, which uses 21-bit T and B fields. CC enables capabilities to be stored in the register file in compressed format, often requiring decoding before use. As a result, this comparison lies on several critical paths in our processor prototype.

The bounds t and b are computed relative to A_{upper}:

$$t = \{(A_{\text{upper}} + c_{\text{t}}), T, zeros\text{'}E\}$$
$$b = \{(A_{\text{upper}} + c_{\text{b}}), B, zeros\text{'}E\}$$
$$\text{where } A_{\text{upper}} = a[31:E+9]$$

The bounds check during memory access is then:

$$b \leqslant computed_address < t$$

In summary, CC generalizes Low-fat arithmetic to allow full use of the power-of-two-sized encoding space for representing addresses outside of the bounds, while improving speed of decoding.

Encoding full address space

The largest encodable 32-bit value of t is `0xFF800000`, making a portion of the address space inaccessible to the largest capability. We can resolve this by allowing t to be a 33-bit value, but this bit-size mismatch introduces some additional complication when decoding t. The following condition is required to correct t for capabilities whose representable region wraps the edge of the address space:

$$if\ ((E < 24)\,\&((t[32:31] - b[31]) > 1))\ then\ t[32] = !t[32]$$

That is, if the length of the capability is larger than E allows, invert the most significant bit of t.

A.3 Fast representable limit checking

Pointer arithmetic is typically performed using addition, and does not raise an exception. If we wish to preserve these semantics for capabilities, capability pointer addition must fit comfortably within the delay of simple arithmetic in the pipeline, and should not introduce the possibility of an exception. For CC, as with Low-fat, typical pointer addition requires adding only an offset to the pointer address, leaving the rest of the capability fields unchanged. However, it is possible that the address could pass either the upper or the lower limits of the representable space, beyond which the original bounds can no longer be reconstituted. In this case, CC clears the tag of the resulting capability to maintain memory safety, preventing an illegal reference to memory from being forged. This check against the representable limit, R, has been designed to be much faster than a precise bounds check, thereby eliminating the costly measures the Low-fat design required to achieve reasonable performance.

To ensure that the critical path is not unduly lengthened, CC verifies that an increment i will not compromise the encoding by inspecting only i and the original address field. We first ascertain if i is *inRange*, and then if it is *inLimit*. The *inRange* test determines whether the magnitude of i is greater than that of the size of the representable space, s, which would certainly take the address out of representable limits:

$$inRange = -s < i < s$$

The *inLimit* test assumes the success of the *inRange* test, and determines whether the update to A_{mid} could take it beyond the representable limit, outside the representable space:

$$inLimit = \begin{cases} I_{\text{mid}} < (R - A_{\text{mid}} - 1), & \text{if } i \geqslant 0 \\ I_{\text{mid}} \geqslant (R - A_{\text{mid}}) \text{ and } R \neq A_{\text{mid}}, & \text{if } i < 0 \end{cases}$$

The *inRange* test reduces to a test that all the bits of I_{top} ($i[63 : E+9]$) are the same. The *inLimit* test needs only 9-bit fields ($I_{\text{mid}} = i[E+8, E]$) and the sign of i.

The I_{mid} and A_{mid} used in the *inLimit* test do not include the lower bits of i and a, potentially ignoring a carry in from the lower bits, presenting an *imprecision hazard*. We solve this by conservatively subtracting one from the representable limit when we are incrementing upwards, and by not allowing any subtraction when A_{mid} is equal to R.

One final test is required to ensure that if $E \geqslant 23$, any increment is representable. (If $E = 23$, the representable space, s, encompases the entire address space.) This handles a number of corner cases related to T, B, and A_{mid} describing bits beyond the top of a virtual address. Our final fast *representability* check composes these three tests:

$$representable = (inRange \text{ and } inLimit) \text{ or } (E \geqslant 23)$$

To summarize, the representability check depends only on four 9-bit fields, T, B, A_{mid}, and I_{mid}, and the sign of i. Only I_{mid} must be extracted during execute, as A_{mid} is cached in our register file. This operation is simpler than reconstructing even one full bound. This fast representability check allows us to perform pointer arithmetic on compressed capabilities directly, avoiding decompressing capabilities in the register file that introduces both a dramatically enlarged register file and substantial load-to-use delay.

References

[1] Akritidis, P. 'Cling: A Memory Allocator to Mitigate Dangling Pointers'. In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. Washington, DC: USENIX Association, 2010, pp. 12–12.

[2] *ARMv8-M Memory Protection Unit*. 0200-00. ARM Ltd. Feb. 2017.

[3] Aspencore. *2017 Embedded Markets Study*. 2017. https://m.eet.com/media/1246048/2017-embedded-market-study.pdf.

[4] Barry, R. *Mastering the FreeRTOS Real Time Kernel*. 161204th ed. FreeRTOS.org. Real Time Engineers Ltd. London, Dec. 2016.

[5] Beniamini, G. *Over The Air: Exploiting Broadcom's Wi-Fi Stack*. 2017. https://googleprojectzero.blogspot.co.uk/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html (visited on 15/05/2017).

[6] Beniamini, G. *QSEE privilege escalation vulnerability and exploit*. 2016. https://bits-please.blogspot.co.uk/2016/05/qsee-privilege-escalation-vulnerability.html (visited on 05/02/2016).

[7] *Bluespec SystemVerilog Version 3.8 Reference Guide*. Bluespec, Inc. Waltham, MA, 2004.

[8] Boehm, H. and Weiser, M. 'Garbage Collection in an Uncooperative Environment'. In: *Softw. Pract. Exper.* 18.9 (Sept. 1988), pp. 807–820.

[9] Brasser, F. et al. 'TyTAN: Tiny trust anchor for tiny devices'. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.

[10] Brion. *JavaScript engine internals: NaN-boxing*. 2018. https://brionv.com/log/2018/05/17/javascript-engine-internals-nan-boxing/.

[11] Brown, J. et al. 'A capability representation with embedded address and nearly-exact object bounds'. In: *Project Aries Technical Memo 5* (2000).

[12] Burow, N. et al. 'Control-Flow Integrity: Precision, Security, and Performance'. In: *ACM Comput. Surv.* 50.1 (Apr. 2017), 16:1–16:33.

[13] Carter, N. P., Keckler, S. W. and Dally, W. J. 'Hardware support for fast capability-based addressing'. In: *SIGPLAN Not.* 29.11 (Nov. 1994), pp. 319–327.

[14] Checkoway, S. et al. 'Comprehensive Experimental Analyses of Automotive Attack Surfaces'. In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 6–6.

[15] Chisnall, D. et al. 'Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015, pp. 117–130.

[16] Colwell, R. P., Gehringer, E. F. and Jensen, E. D. 'Performance Effects of Architectural Complexity in the Intel 432'. In: *ACM Trans. Comput. Syst.* 6.3 (Aug. 1988), pp. 296–339.

[17] Costin, A. et al. 'A Large-Scale Analysis of the Security of Embedded Firmwares'. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 95–110.

[18] Cowan, C. et al. 'StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks'. In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, pp. 5–5.

[19] Dang, T. H. Y., Maniatis, P. and Wagner, D. 'Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers'. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 815–832.

[20] Davi, L. et al. 'HAFIX: Hardware-Assisted Flow Integrity eXtension'. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.

[21] Davi, L., Koeberl, P. and Sadeghi, A. R. 'Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation'. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2014, pp. 1–6.

[22] Davis, B. et al. 'CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 379–393.

[23] Dennis, J. B. and Horn, E. C. Van. 'Programming semantics for multiprogrammed computations'. In: *Commun. ACM* 9.3 (1966), pp. 143–155.

[24] Dent, S. *Google and others back Internet of Things security push*. 2017. https://www.engadget.com/2017/10/23/google-arm-internet-of-things-security/ (visited on 21/11/2018).

[25] Dhurjati, D. et al. 'Memory Safety Without Garbage Collection for Embedded Applications'. In: *ACM Trans. Embed. Comput. Syst.* 4.1 (Feb. 2005), pp. 73–111.

[26] Ecker, W., Müller, W. and Dömer, R. *Hardware-dependent Software: Principles and Practice*. 1st. Springer Publishing Company, Incorporated, 2009.

[27] Esswood, L. 'CheriOS: A high-performance and completely untrusted single-address-space capability operating system'. PhD Thesis. Cambridge, UK: University of Cambridge, 2019. to be submitted.

[28] Evans, J. 'A Scalable Concurrent malloc(3) Implementation for FreeBSD'. In: *BSDCan*. 2006.

[29] George, P. et al. 'MIPSpro™ N32 ABI Handbook'. In: (2002).

[30] Gras, B. et al. 'ASLR on the Line: Practical Cache Attacks on the MMU'. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA*. 2017.

[31] Gumpertz, R. H. 'Error Detection with Memory Tags'. PhD thesis. Carnegie Mellon University, 1981.

[32] Guthaus, M. R. et al. 'MiBench: A free, commercially representative embedded benchmark suite'. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14.

[33] Henning, J. L. 'SPEC CPU2006 Benchmark Descriptions'. In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006).

[34] Joannou, A. et al. 'Efficient Tagged Memory'. In: *2017 IEEE International Conference on Computer Design (ICCD)*. Nov. 2017, pp. 641–648.

[35] Joannou, A. J. P. 'High-performance memory safety - Optimizing the CHERI capability machine'. PhD thesis. University of Cambridge, Computer Laboratory, May 2018.

[36] Klein, G. et al. 'seL4: Formal Verification of an OS Kernel'. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220.

[37] Koeberl, P. et al. 'TrustLite: A Security Architecture for Tiny Embedded Devices'. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 10:1–10:14.

[38] Kouwe, E. van der, Nigade, V. and Giuffrida, C. 'DangSan: Scalable Use-after-free Detection'. In: *EuroSys*. 2017.

[39] Kwon, A. et al. 'Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security'. In: *20th Conference on Computer and Communications Security*. ACM, 2013.

[40] Lea, D. *A Memory Allocator*. 2000.
http://g.oswego.edu/dl/html/malloc.html.

[41] Levy, H. M. *Capability-Based Computer Systems*. Digital Press, 1984.

[42] Lindqvist, U. and Neumann, P. G. 'The Future of the Internet of Things'. In: *Commun. ACM* 60.2 (Jan. 2017), pp. 26–30.

[43] Liu, D., Zhang, M. and Wang, H. 'A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping'. In: *CCS*. 2018.

[44] Lougher, R. *JamVM*. 2014.
http://jamvm.sourceforge.net/.

[45] Midi, D., Payer, M. and Bertino, E. 'Memory Safety for Embedded Devices with nesCheck'. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. Abu Dhabi, United Arab Emirates: ACM, 2017, pp. 127–139.

[46] Nagarakatte, S. et al. ‘SoftBound: Highly Compatible and Complete Spatial Memory Safety for C’. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: ACM, 2009, pp. 245–258.

[47] Needham, R. M. and Walker, R. D. H. ‘The Cambridge CAP computer and its protection system’. In: *Proceedings of the sixth ACM symposium on Operating systems principles*. SOSP ’77. West Lafayette, Indiana, United States: ACM, 1977, pp. 1–10.

[48] Noorman, J. et al. ‘Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base’. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 479–498.

[49] Oleksenko, O. et al. ‘Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack’. In: *Proc. ACM Meas. Anal. Comput. Syst.* 2.2 (June 2018), 28:1–28:30.

[50] *Oracle’s SPARC T7 and SPARC M7 Server Architecture*. Oracle. Aug. 2016.

[51] Parkinson, M. et al. *Project Snowflake: Non-blocking safe manual memory management in .NET*. Tech. rep. 2017.

[52] Payer, M. ‘Too much PIE is bad for performance’. In: (2012).

[53] Peleg, O. et al. ‘Utilizing the IOMMU Scalably’. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 549–562.

[54] Saltzer, J. H. and Schroeder, M. D. ‘The Protection of Information in Computer Systems’. In: *Communications of the ACM* 17.7 (July 1974).

[55] Schöne, R. et al. ‘Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance’. In: *CoRR* abs/1905.12468 (2019). arXiv: 1905.12468.

[56] Serebryany, K. et al. ‘AddressSanitizer: A Fast Address Sanity Checker’. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC’12. Boston, MA: USENIX Association, 2012, pp. 28–28.

[57] Serebryany, K. et al. ‘Memory Tagging and how it improves C/C++ memory safety’. In: *CoRR* abs/1802.09517 (2018). arXiv: 1802.09517.

[58] Silvestro, S. et al. ‘FreeGuard: A Faster Secure Heap Allocator’. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, 2017, pp. 2389–2403.

[59] Soares, L. and Stumm, M. ‘Exception-less System Calls for Event-driven Servers’. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’11. Portland, OR: USENIX Association, 2011, pp. 10–10.

[60] Soares, L. and Stumm, M. ‘FlexSC: Flexible System Call Scheduling with Exception-less System Calls’. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 33–46.

[61] Strackx, R., Piessens, F. and Preneel, B. ‘Efficient Isolation of Trusted Subsystems in Embedded Systems’. In: *Security and Privacy in Communication Networks*. Ed. by Jajodia, Sushil and Zhou, Jianying. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 344–361.

[62] Szekeres, L. et al. ‘SoK: Eternal War in Memory’. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62.

[63] Thomas, G. *A proactive approach to more secure code*. 2019. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (visited on 05/11/2019).

[64] *TrustZone technology for ARMv8-M Architecture*. 0101-00. ARM Ltd. Aug. 2016.

[65] Ukil, A., Sen, J. and Koilakonda, S. ‘Embedded security for Internet of Things’. In: *2011 2nd National Conference on Emerging Trends and Applications in Computer Science*. 2011, pp. 1–6.

[66] Waterman, A. et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. Tech. rep. EECS Department, University of California, Berkeley, 2017.

[67] Watson, R. N. et al. ‘Fast Protection-Domain Crossing in the CHERI Capability-System Architecture’. In: *IEEE Micro* 36.5 (2016), pp. 38–49.

[68] Watson, R. N. M. et al. ‘A Taste of Capsicum: Practical Capabilities for UNIX’. In: *Commun. ACM* 55.3 (Mar. 2012), pp. 97–104.

[69] Watson, R. N. M. et al. *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide*. Tech. rep. UCAM-CL-TR-877. University of Cambridge, Computer Laboratory, Sept. 2015.

[70] Watson, R. N. M. et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)*. Tech. rep. UCAM-CL-TR-907. University of Cambridge, Computer Laboratory, Apr. 2017.

[71] Watson, R. N. M. et al. 'CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization'. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 20–37.

[72] WikiDevi. *a user-editable database for computer hardware based on MediaWiki and Semantic MediaWiki*. https://wikidevi.com/wiki/Main_Page.

[73] Wilkes, M. V. *The Cambridge CAP Computer and Its Operating System (Operating and Programming Systems Series)*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1979.

[74] Woodruff, J. et al. 'CHERI Concentrate: Practical Compressed Capabilities'. In: *IEEE Transactions on Computers* (2019).

[75] Woodruff, J. et al. 'The CHERI Capability Model: Revisiting RISC in an Age of Risk'. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468.

[76] Wulf, W. et al. 'HYDRA: The Kernel of a Multiprocessor Operating System'. In: *Commun. ACM* 17.6 (June 1974), pp. 337–345.

[77] Xia, H. et al. 'CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety'. In: *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: ACM, 2019, pp. 545–557.

[78] Xia, H. et al. 'CheriRTOS: A Capability Model for Embedded Devices'. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018, pp. 92–99.