Technical Report

UCAM-CL-TR-975
ISSN 1476-2986

Number 975

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety

Brett Gutstein

November 2022

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

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

This technical report is based on a dissertation submitted July 2022 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

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

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

ISSN 1476-2986

Abstract

Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety

Brett Ferdosi Gutstein

CHERI (Capability Hardware Enhanced RISC Instructions) is a promising research processor-architecture protection model that facilitates memory safety and fine-grained compartmentalization for software. The architecture has reached a mature state and been integrated into Arm's industrial-scale Morello system-on-chip, a large corpus of software has been adapted to support CHERI, and prior work has demonstrated that replacing integer pointers with CHERI capabilities can make C and C++ programs spatially safe. In this dissertation, I identify gaps that limit the ability of current mitigations based on CHERI to deliver real-world vulnerability protection, and I work towards addressing them.

I develop the memory-operations framework (MOF) for reasoning about memory-safety mitigations and the types of attacks they prevent. I apply the MOF to analyze CheriABI, the most sophisticated memory-safety mitigation built atop CHERI. I also evaluate CheriABI's effectiveness in mitigating a set of real-world attacks that targeted devices running Apple's iOS. Based on this evaluation, I identify two key areas in CHERI-supported memory safety that require improved protections.

One of these areas involves support for contemporary programming language interpreters, which have not previously been adapted to CHERI. Using Apple's JavaScriptCore as a case study, I evaluate the feasibility, source-code compatibility, and security properties of adapting an interpreter that supports just-in-time compilation to CHERI. I determine that such an adaptation is feasible, practical, and can achieve parity with more typical applications in terms of memory protection.

The other area is providing temporal safety for userspace heaps, which CheriABI does not currently support. I introduce novel algorithms and software components that constitute a fully elaborated system for CHERI-based userspace heap temporal safety. I implement the system, which includes the Cornucopia kernel subsystem for sweeping capability revocation and a generic userspace library that encapsulates changes required for memory allocators, in CheriBSD for Morello. Relative to the CHERIvoke algorithm for heap temporal safety, which has previously been published but not implemented on CHERI hardware, the novel algorithms reduce application runtimes by up to 23.5% and pause times by up to 11,000x, potentially making temporal safety with CHERI feasible for large, real-world workloads.

Acknowledgements

Thanks to my supervisor Robert Watson for his guidance, understanding, and support. To my colleague Wes Filardo for his kindness and erudition. To all members of the CHERI project, and especially Alex Richardson, Jessica Clarke, Peter Neumann, and John Baldwin, for making this work possible. To my examiners Alan Mycroft and Howie Shrobe for their valuable engagement. Thanks to the Gates Cambridge Trust for the generous funding, and to Arm Limited for providing access to computing resources. Above all, thanks to my family and friends, without whom I would not have made it to the finish line.

Contents

Chapter 1

Introduction

Computers govern increasingly important aspects of our lives, but security for computer systems remains elusive. Cybersecurity breaches in major organizations, including computing giants like Facebook [16], fuel pipeline operators [80], and the United Kingdom's National Health Service [117], regularly affect millions of people, causing significant financial and human losses.

The security vulnerabilities that facilitate these breaches can be divided into three types. The first type involves problems related to human-computer use. A computer system that is perfectly engineered can still be attacked if the people using it are vulnerable to manipulation. For example, they might have their credentials stolen or be influenced to leak sensitive data. A July 2020 Twitter attack that compromised 130 of the highest profile accounts and used them to solicit cryptocurrency transfers was carried out by a 17-year-old who used social engineering to access an internal Twitter tool [114]. While defending against these kinds of attacks is crucially important, the multidisciplinary effort required to do so is beyond the scope of this dissertation.

The second type involves logic errors in the engineering of computer systems. For example, a programmer might forget to include the appropriate authentication checks before performing some privileged operation. These types of errors might seem unlikely to pass rigorous testing and quality assurance protocols, but obvious mistakes end up in even the most widely used systems. A bug in an Uber API, discovered in 2019, allowed any user to obtain sensitive information and take over the account of any other user for whom they had a phone number or email address [123]. The flaw behind this bug was a failure to authenticate the user making the request. It is impossible to detect logic errors automatically in general, since any automatic checker must be given a correct abstract specification of the system that is itself free from logic errors. Logic errors are also out of scope for this dissertation.

The third type concerns vulnerabilities in the engineering of computer systems that can be detected and mitigated automatically. Memory-safety vulnerabilities, which are possible in low-level languages like C and C++ when a program allows for the integrity of its memory objects to be violated, fall into this category. There are various techniques for mitigating memory-safety vulnerabilities automatically, but so far none has achieved widespread adoption. In 2019, Microsoft reported that memory-safety errors are behind approximately 70% of security vulnerabilities in their products [22], and in 2016 Google stated that about 86% of Android vulnerabilities are memory-safety related [23].

One way to mitigate memory-safety vulnerabilities automatically is to use higher-level languages that eliminate them by design. However, it would require a huge amount of

labor and expertise to rewrite and retest the hundreds of millions of lines of low-level code that underpin current computing infrastructure. Furthermore, the loss of fine-grained system control that comes with higher-level languages limits their suitability for certain kinds of software.

It is also possible to mitigate memory-safety vulnerabilities automatically for existing C/C++ codebases by augmenting compilers with instrumentation and dynamic checks. To date, however, schemes that do so in software have come with high performance overheads that make them impractical. Hardware support has the potential to reduce overheads significantly, but past hardware designs that support the effective mitigation of memory-safety vulnerabilities have not been deployed in realistic, high-fidelity microarchitectures.

CHERI (Capability Hardware Enhanced RISC Instructions) [118, 120, 125], is a promising research processor-architecture protection model that facilitates memory safety for C and C++ codebases by extending existing instruction set architectures with hardware-tagged memory capabilities. Over more than ten years of research effort, the CHERI project has reached a mature state, and a large corpus of software has been adapted to support it. This corpus includes the CHERI-LLVM compiler [27], the general-purpose CheriBSD operating system [25], special-purpose operating systems like CheriFreeRTOS [26], and a number of substantive applications such as PostgreSQL [28]. Currently, the most sophisticated memory-safety mitigation built using CHERI is the CheriABI C/C++ execution environment, which has been shown to provide complete spatial memory safety for C/C++ programs at a reasonable performance cost [32, 99].

CHERI technology has been deployed and validated in increasingly high-fidelity systems. CHERI extensions exist for MIPS, RISC-V, and ARM architectures, and example processor implementations have been developed for QEMU and FPGA. The ARMv8-A extension, known as Morello, was developed by the University of Cambridge in collaboration with Arm and is the subject of the £187M UKRI Digital Security by Design program [69, 70, 113]. As part of this program, Arm has produced an industrial-quality system-on-chip, with superscalar, out-of-order CPU cores, that implements the CHERI Morello architecture. The system-on-chip and the architecture it implements are both called Morello.

Despite impressive results so far, there remain various outstanding research questions that must be addressed to facilitate robust memory safety and real-world vulnerability mitigation with CHERI. In the following chapters, I identify some of these gaps and work towards closing them.

1.1 Contributions

In this dissertation, I make a number of contributions related to analyzing and extending the security properties that C/C++ applications can achieve with CHERI-supported memory safety:

Security analysis I develop the novel memory-operations framework (MOF) for reasoning about memory-safety mitigations and the types of attacks that they prevent. Using the MOF, I systematize the security properties of CheriABI [32], which has not previously been done. I further use the framework to analyze exploits that were carried out against devices running Apple's iOS in the wild to determine how CheriABI memory-safety protections would fare against contemporary real-world attacks. Based on this analysis, I identify important gaps in current research on

memory safety with CHERI; namely, support for CHERI in programming-language interpreters and temporal safety for userspace heaps.

Programming-language interpreters I perform a feasibility study to determine how CHERI memory-safety protections can be applied to harden contemporary interpreters that feature just-in-time compilation. Such interpreters have different characteristics from typical applications, and they have not been investigated in previous work on CHERI-supported memory safety. I adapt JavaScriptCore, the JavaScript interpreter in Apple's WebKit browser framework, to compile as pure-capability code, i.e. so that all pointers are implemented as CHERI capabilities. This adaptation includes a just-in-time compiler targeting the Morello platform. I determine that it is indeed possible to perform such an adaptation without affecting the interpreter's functionality, and that the source-code compatibility burden of doing so is comparable to that of adapting similar types of software. Applying the MOF, I determine that, unlike typical applications, language interpreters do not automatically gain all CheriABI security properties when adapted to support pure-capability compilation. However, I discover that straightforward changes to their memory allocators, JIT compilers, and build systems can bring their security properties up to par.

Userspace heap temporal safety I propose, implement, and evaluate novel techniques and software components that constitute a fully elaborated system for CHERI-based userspace heap temporal safety. I introduce Cornucopia [122], a novel kernel subsystem of the CheriBSD operating system that userspace heap allocators can utilize to guarantee temporal safety. Cornucopia performs sweeping capability revocation for userspace processes. It can be configured to use the previously published CHERIvoke algorithm [127], which has never before been implemented on real CHERI hardware, as well as two novel algorithms with improved performance characteristics. These novel algorithms are inspired by past work on garbage collection and use Morello's architectural features to implement store-side or load-side barriers that allow sweeping capability revocation to be performed concurrently with the running application.

I evaluate these algorithms on Morello, producing the first performance results from a superscalar ASIC-based CHERI capability machine to be published. I determine that, relative to the CHERIvoke baseline, the novel algorithms reduce application runtimes by up to 23.5% and pause times by up to 11,000x. For the load-side barrier algorithm, application pause times are constant and do not scale with memory footprint; this feature potentially makes sweeping capability revocation feasible for large, real-world workloads.

I systematize a set of requirements that userspace heap allocators must meet in order to leverage the Cornucopia kernel subsystem to provide heap temporal safety, and I describe techniques for meeting them. I evaluate the effectiveness of heap temporal safety with Cornucopia and determine that it is possible to implement sweeping capability revocation on real CHERI hardware to mitigate heap temporal safety vulnerabilities in practice.

I also introduce a generic wrapper library that allows allocators to implement heap temporal safety with Cornucopia at a low source-code compatibility cost. This library demonstrates the unexpected result that it is possible to encapsulate temporal-safety

logic without access to an allocator's internal data structures and synchronization primitives. Relative to modifying a contemporary memory allocator to integrate with Cornucopia directly, using this library can require fewer than 1% of the source-code changes and incur a performance cost of less than 5% in terms of runtime, CPU utilization, and DRAM traffic.

1.2 Publications

During my PhD, I co-authored and delivered the following papers and presentations:

Peer-reviewed conference papers

- Nathaniel Wesley Filardo, Brett F. Gutstein, *et al. Cornucopia: Temporal safety for CHERI heaps.* In the 2020 IEEE Symposium on Security and Privacy (SP).
- A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, *et al. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals.* In Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS).

Invited papers

- Brett F. Gutstein *et al. The ECATS project: Developments in the CHERI security architecture.* In Government Microcircuit Applications and Critical Technology Conference (GOMACTech) 2021.
- Brett F. Gutstein *et al. The ECATS project: CHERI and Thunderclap.* In Government Microcircuit Applications and Critical Technology Conference (GOMACTech) 2020.

Peer-reviewed presentations

- Brett F. Gutstein. *Challenges in system security evaluation.* At UK Systems Research Challenges Workshop 2019, Northumberland, UK.

Invited keynote presentations

- Brett F. Gutstein. *Thunderclap: a novel threat from malicious peripheral devices.* At NASK Secure Early Bird 2019, Warsaw, Poland.

Reports

- Peter G. Neumann and Brett F. Gutstein. *CHERI's mitigation of security vulnerabilities.* Produced as part of the DARPA SSITH ECATS project.
- Nathaniel Wesley Filardo, Brett F. Gutstein, *et al. Achelous: whence a better Cornucopia?.* Technical report, in preparation.

1.3 Infrastructural contributions

I also made a number of scientific and engineering contributions to infrastructure that will support further CHERI-related research.

- I contributed to validation of the Arm Morello platform. I identified and reported issues in the Morello specification and carried out numerous tests on a high-fidelity FPGA implementation of the Morello hardware. One of these tests uncovered a hardware issue related to tag memory aliasing that was able to be worked around in firmware.

- I assisted in bringing up the CheriBSD operating system on Morello. I contributed kernel modifications for capability-aware boot, context switching, signal handling, and process startup. I also added initial runtime linker support for dynamically linked CheriABI binaries and modified a number of userspace applications to build correctly for Morello. I fixed multiple bugs in the `hwpmc` performance monitoring framework and added support for profiling position-independent executables.

- I adapted the JavaScriptCore JavaScript runtime, which is part of the WebKit framework that powers browsers including Apple's Safari and Mobile Safari, to Morello. This adaptation included support for multiple tiers of JavaScript execution, including the baseline JIT compiler, which was adapted in collaboration with Arm Limited.

- I adapted machine-dependent parts of CheriBSD's capability-revocation kernel subsystem to Morello.

- I designed and implemented a generic wrapper for userspace allocators that allows them to make use of sweeping capability revocation with minimal source code modification.

1.4 Work performed in collaboration with others

The CHERI project is a large and highly collaborative effort that comprises over a decade of work by dozens of team members. The work in this dissertation builds indirectly and directly on past contributions by others. I performed all of the experiments described, and, unless explicitly stated otherwise, I developed the ideas and systems under test, except for the following work performed by and in collaboration with others:

- Wes Filardo led development of the load-side and store-side barrier algorithms for concurrent sweeping capability revocation. I reviewed the store-side algorithm and helped to design the load-side algorithm. The evaluation of these algorithms in this dissertation is novel and solely my work.

  Wes implemented the machine-independent, MIPS, and RISC-V components of the Cornucopia kernel subsystem for sweeping revocation in CheriBSD. I reviewed and made fixes to these components. I added a machine-dependent component for Morello, which leveraged features including hardware capability-dirty-bit tracking that are not present in other architectures.

Wes added direct integration with Cornucopia to the `snmalloc` memory allocator. This allocator configuration is one of many that I use in my performance evaluation.

- Khilan Gudka and Alex Richardson initially adapted QtWebkit, including the C++ backend of JavaScriptCore's Low Level Interpreter, to support CHERI. Ruben Aryapetyan later extended their work to support JavaScriptCore up to the Baseline JIT compiler, targeting a precursor of Morello. I adapted the extension to target Morello, added features, and made a number of fixes and cleanups. Jacob Bramley helped to shepherd the Morello adaptation for public release.

- Brooks Davis and Hongyan Xia added direct integration with Cornucopia to the `dlmalloc` memory allocator. This allocator configuration is one of many that I use in my performance evaluation.

1.5 Outline

The rest of this dissertation is structured as follows. Chapter 2 covers background information that sets the stage for later chapters. In Chapter 3, I develop the memory-operations framework, as previously described in Section 1.1. I apply the MOF to analyze CheriABI and real-world exploits carried out against systems running iOS. Based on this analysis, I identify gaps in current CheriABI protections. These gaps include programming-language interpreters and temporal safety, as previously described in Section 1.1, and they motivate my contributions in Chapters 4 and 5. Chapter 4 investigates how contemporary language interpreters can be adapted to support CHERI. Chapter 5 covers novel sweeping capability revocation algorithms, the fully elaborated Cornucopia kernel subsystem for capability revocation, and the generic userspace library that allows heap allocators to offer temporal safety with little source code modification. Chapter 6 concludes the dissertation.

Chapter 2

Background

In this chapter, I cover background information and related work relevant to the rest of the dissertation. I first introduce the concept of memory safety in languages like C and C++, explain how memory safety bugs can be exploited, and summarize past memory safety attack techniques and mitigations. I then describe CHERI in detail, with a specific focus on Arm's Morello platform. I highlight CHERI properties that can help address problems related to memory safety, and I review previous work on C/C++ memory safety with CHERI. I then discuss virtual-machine language interpreters, operating-system virtual memory, and garbage collectors, which are relevant to work described in later chapters.

2.1 C/C++ memory safety

Generally speaking, memory safety is a property of individual programs. It describes whether or not a program allows the integrity of its memory objects to be violated during execution.

Memory objects are memory-backed constructs such as data structures and primitive data types. Their exact size and layout is determined by the program's compiler, and they might be allocated on the stack, in the heap, or as globals. Memory objects have abstract bounds that are determined by their size and location in memory as well as abstract lifetimes that are determined by their storage class.

Memory safety can be divided into three aspects:

Spatial safety A program is spatially safe if it guarantees the integrity of memory object bounds. This means that a reference to a memory object cannot be used to access memory outside of that object, e.g. by over-running a buffer contained in the memory object.

Temporal safety A program is temporally safe if it guarantees the integrity of memory object lifetimes. This means that memory used to store one memory object cannot be used simultaneously to store another memory object, e.g. as a result of dangling pointers to the heap.

Type safety A program is type safe if it guarantees the integrity of memory object types. This means that a memory object referenced as one type cannot simultaneously be referenced as a memory object of an incompatible type, e.g. via a C-style union or type cast.

Some programming languages, such as Java and OCaml, produce programs that are memory-safe by construction. They do not allow the programmer to express operations that would violate the integrity of memory objects.

On the other hand, languages like C and C++ give programmers significant control over data representation and pointer use. This control makes C and C++ well suited to systems programming, but it also allows for the creation of programs that are not memory safe.

Features of a program that allow for the integrity of memory objects to be violated are known as memory-safety bugs. Memory-safety bugs can be related to spatial safety, temporal safety, or type safety. For example, if a program allocates a buffer and allows for writing past the end of it, then the program contains a spatial-safety bug. Similarly, if a program allows for a heap memory object to be accessed after it has been freed, then it contains a temporal-safety bug.

The C standard does specify the representations (Section 6.2.6), lifetimes (Section 6.2.4), and compatible types (Section 6.2.7) of memory objects [95]. However, corresponding violations of memory object bounds, lifetimes, and types are left as undefined behavior. This means that a C program is allowed to take any action in the presence of a memory-safety bug. Furthermore, recent work on C semantics by Memarian *et al.* identified areas related to memory-safety bugs in which the standard's current specification is unclear or inadequate [81].

In practice, mainstream compilers handle memory-safety bugs in ways that can make C/C++ programs susceptible to attack. When memory-safety bugs make a program susceptible to attack, they can also be called memory-safety vulnerabilities. For example, the infamous 1988 Morris worm leveraged a buffer overflow spatial-safety vulnerability in the `fingerd` program to increase its privilege on targeted hosts [107]. The behavior of mainstream compilers constitutes a murky, unpublished *de facto* standard that developers have grown to depend on for program functionality [82].

It is theoretically possible to write C/C++ programs that are memory-safe when built by any compiler that conforms to the published standards [95, 126]. Specifically, C/C++ programs that do not contain any kind of memory-safety bug or other undefined behavior are memory safe by definition. Unfortunately, however, the continued prevalence of security incidents caused by memory-unsafe C/C++ programs [22, 23] suggests that producing memory-safe C/C++ programs is exceedingly difficult.

In the early 2000s, Morrisett *et al.* created a research memory-safe dialect of C called Cyclone [83]. It modifies the C-language syntax and semantics in order to make some kinds of memory-safety bugs impossible to express. Other kinds are prevented by dynamic checks. This means that they are not exploitable, i.e. they are not memory-safety vulnerabilities.

It is also possible to specify a syntactically and semantically compatible memory-safe version of C/C++ that refines aspects of the current standards' undefined behavior such that all memory-safety bugs must not be exploitable. The recently developed CHERI C [81], which specifies that a program should trap when certain aspects of memory-safety are violated at runtime, makes significant progress in this direction. Such a language could be implemented on current mainstream hardware by augmenting a C/C++ compiler to emit instrumentation and dynamic checks. To date, however, no mainstream implementation of C/C++ has been made memory-safe, likely because the performance cost of enforcing memory safety with existing hardware is prohibitive [106]. Capability hardware like CHERI is a promising potential solution to this problem, but, until it is suitable for widespread

deployment, a huge quantity of C/C++ programs will remain susceptible to memory-safety attacks.

2.1.1 Attacks and limited mitigations

In a memory-safety attack, also known as a memory-safety exploit, an attacker leverages a memory-safety vulnerability in some program to gain privilege that they were not intended to have. The usual goal of this privilege escalation is to execute arbitrary code in the target process, but other goals, such as the exfiltration of sensitive data, are also common. Memory-safety attacks have been prevalent, and extremely costly, throughout the history of computing, and a large number of mitigations has been developed in an attempt to stop them. A memory-safety mitigation alters the execution environment of a program to make one or more memory-safety attacks unsuccessful. Mitigations can aim to prevent specific kinds of attacks from succeeding, or, more powerfully, to render entire classes of memory-safety vulnerability unexploitable.

Here I present a brief history of widely deployed memory-safety attacks and mitigations. It highlights the whack-a-mole nature of mitigations in the real world to date. When a new attack becomes popular, a mitigation for the attack is developed and deployed. However, the mitigation fails to address the attack's underlying vulnerability, so a variant of the attack that bypasses the mitigation pops up, and the process repeats.

Strictly speaking, a memory-safety attack or exploit is a particular sequence of actions carried out by an attacker to take advantage of a memory-safety vulnerability in a particular program. However, in the following discussion, I refer abstractly to groups of similar memory-safety attacks.

Attack: code injection Perhaps one of the earliest known memory-safety attack techniques involves exploiting buffer overflows on the stack to inject and execute attacker-supplied code [90]. A programmer might, for example, allocate a buffer on the stack to hold some string argument provided by the user of a program. If the user-provided string argument is larger than the allocated buffer, and if the programmer uses a function like `strcpy()`, which copies the argument until it reaches a null terminator, then the argument will be copied past the bounds of the buffer. A malicious user could exploit such a program with a code-injection attack by supplying a specially formatted string. The string might contain a block of arbitrary code and a carefully positioned pointer, which overwrites the current function's return address on the stack with the address of the code block. Then when the function returns, the arbitrary attacker-supplied code is run.

Mitigation: write xor execute A widely employed mitigation that prevents the above type of stack-smashing attack is write xor execute (W⊕X) protection, also known as data execution prevention (DEP) [96]. This mitigation enforces the invariant that pages of a process' memory can be mapped with either write permission or execute permission, but not both, except in special circumstances. The effect of W⊕X protection is that, in general, an attacker cannot inject arbitrary code and then induce the process to run it. Some exceptions exist for programs like just-in-time (JIT) compilers, which write code that later becomes executable as part of their normal operation.

Attack: code reuse Not easily deterred, attackers developed new techniques to bypass W⊕X protection: code-reuse attacks. The idea behind these attacks is to chain together snippets of existing library code, called gadgets, in a way that ultimately grants the attacker arbitrary code execution. The most widely known code reuse technique is return-oriented programming (ROP) [100], in which gadgets typically end with a return instruction, but other variants are possible [12, 20]. In the example above, a ROP attacker would use the buffer overflow to carefully place data and ROP gadget addresses on the stack, so that they appear to be function arguments and return addresses. The current function's return address would be overwritten with the known address of the initial ROP gadget. Then, when the function returns, the ROP gadget chain created on the stack is executed, which allows the attacker to execute arbitrary code.

Mitigations: stack canaries and shadow stacks Stack canaries and shadow stacks were developed to make it more difficult to overwrite a return address stored on the stack [31]. In a stack canary scheme, a value known as the canary is stored as part of stack frames, between local storage and the function's return address. All function epilogues check that the canary's value matches what was initially stored there before returning. Assuming the canary's value is unknown to the attacker, this mitigation prevents contiguous stack buffer overflows with high probability. However, stack canaries fail to mitigate bugs that allow targeted memory writes on the stack; these targeted writes could overwrite a return address without touching the canary.

Shadow stack schemes involve maintaining a second stack onto which function prologues store return addresses. Function epilogues then compare the shadow stack return address with that of the main stack, detecting an attack if the two do not match. These schemes protect against all attacks that overwrite stack-based return addresses. However, they do not mitigate bugs that allow for the corruption of other code pointers, such as virtual function table pointers in C++ objects.

Attack: beyond the stack With a number of mitigations deployed to make stack exploitation more difficult, attackers found other sources of vulnerabilities. In particular, memory objects on the heap often contain code pointers, which can be overwritten to carry out code-reuse attacks using special gadgets that pivot the stack to memory that the attacker controls. A spatial vulnerability related to the field of some heap memory object can allow attackers to overwrite code pointers in other objects on the heap, or even within another field of the same object. Similarly, temporal safety vulnerabilities related to a heap memory object can allow attackers to overwrite code pointers in another aliased memory object.

Mitigation: Address space layout randomization Address space layout randomization (ASLR) is a widely deployed mitigation against code-reuse attacks. It involves randomizing process memory layouts so that attackers overwriting a code pointer cannot reliably target specific gadgets. However, the probabilistic protection provided by ASLR can be bypassed in a number of ways, including via leaks that reveal the process' memory layout and via system side-channels [46, 60].

Mitigation: Control-flow integrity Some mitigation techniques are designed to prevent code-reuse attacks by protecting a program's control-flow integrity (CFI) [1, 24].

They aim to ensure that branches taken at runtime match a predetermined control-flow graph for the program, or do not violate static type information, usually by performing software instrumentation. However, these kinds of mitigation often provide provide coarse-grained or incomplete protection that can be bypassed easily.

Attack: Data-oriented programming Attention has also turned to data-oriented programming (DOP) attacks, which involve exploiting memory safety bugs without overwriting any pointers. Hu *et al.* demonstrated that, corrupting only data fields, it is theoretically possible to construct expressive, even Turing-complete, exploits against a wide variety of programs [51]. Proposed mitigations for these attacks include enforcing data-flow integrity [19], which is analogous to CFI for data, and detecting the effects of DOP on control-flow tracing [21].

2.1.2 More robust mitigations

Here I describe past research efforts to add more robust memory-safety mitigations to C/C++ implementations. The limited mitigations described above, which generally aim to prevent a single type of exploit, have been sidestepped by attackers relatively easily. These mitigations, on the other hand, attempt to eliminate entire classes of vulnerability, i.e. spatial-safety, temporal-safety, or type-safety vulnerabilities, that facilitate memory safety attacks. For the most part, they have failed to achieve widespread deployment because of performance and compatibility overheads, or because they fail to provide complete protection in practice.

2.1.2.1 Spatial safety

Some past projects have focused on providing spatial safety for C/C++:

Bounded-object approaches Some approaches for spatially safe C have involved software instrumentation that tracks bounds information for each memory object [35, 58, 101]. However, these approaches require that all pointers to a particular object share the same bounds, and they have difficulty enforcing sub-object bounds completely [84].

Fat-pointer approaches A more effective approach is to track bounds information for each pointer rather than each memory object. Pointers with associated bounds information are known as fat pointers. SoftBound [84] is a source-code compatible and effective fat-pointer mitigation based on software instrumentation. It stores bounds information for pointers disjointly in tables that are checked before each pointer dereference. However, SoftBound's spatial safety comes with a prohibitive average runtime overhead of 67%. Other similar software techniques have attempted to trade off mitigation effectiveness for improved performance [50].

Some past work has also proposed hardware support for bounding pointers to provide spatial safety. The M-Machine [18], SAFE low-fat pointers [62], and HardBound [34] feature pointers whose bounds are maintained and tagged by hardware. Hardware support facilitates a reduced performance overhead, but novel hardware comes with significant production and adoption costs.

2.1.2.2 Heap temporal safety

Various past projects provide temporal safety for the C/C++ heap:

Lock-and-key instrumentation One approach for providing heap temporal safety is to maintain a unique identifier for each heap memory object. When a heap pointer is created, it is associated with the identifier of its corresponding heap object. Then, when it is dereferenced, the system checks whether the pointer's identifier still matches that of the referenced memory. This technique is known as lock-and-key mitigation. CETS [85] is a mitigation that uses compiler instrumentation to implement lock-and-key checking. It comes with a runtime overhead of 48% on average. When combined with a spatial safety mitigation, which is a prerequisite for temporal safety with CETS, this overhead jumps to 116%.

Garbage collectors Another approach for heap temporal safety is to use a garbage-collected heap rather than relying on manual calls to `free()`. The Boehm-Demers-Weiser garbage collector [14] is a conservative mark-and-sweep garbage collector for C and C++. MarkUs [2] offers improved security properties by only collecting garbage that has been explicitly freed by the programmer. CRCount [105] is a reference-counting garbage collector that uses static analysis and instrumentation of pointer-related operations to identify pointers to objects in the heap; it associates a reference count with each heap object that gets modified as pointers are created and destroyed. I discuss garbage collectors in more detail in Section 2.8.

Pointer nullification It is also possible to provide heap temporal safety by rendering pointers to freed heap memory unusable. This is known as pointer nullification or invalidation. DANGNULL [64] is a pointer nullification system that uses software instrumentation and runtime data structures to track the pointers that reference heap memory objects. When a heap memory object is freed, pointers referencing it are set to null. DangSan [61] and pSweeper [76] are higher-performance invalidation systems that support concurrency.

Virtual memory protection Electric Fence [41], Dhurjati and Adve [36], and more recently the Oscar system [30], use virtual memory to provide heap temporal safety in a way that does not require software instrumentation to identify pointers: each allocated memory object is placed in a separate virtual page that is rendered inaccessible by MMU hardware once the object is freed. However, this kind of technique can result in memory fragmentation and TLB pressure. I discuss virtual memory further in Section 2.7.

There are also some software-based techniques that reduce the possibility of exploiting a heap temporal-safety vulnerability but do not eliminate it. DieHard [11] provides probabilistic protection by randomizing the location of objects on the heap. Cling [3] is a memory allocator that restricts memory reuse to objects of a similar layout, which makes attacks that exploit heap temporal safety vulnerabilities harder to carry out but not impossible.

2.1.2.3 Complete memory safety

Other projects aim to provide spatial, temporal, and type safety for C/C++:

C dialects Some approaches have attempted to offer complete memory safety without hardware support by making changes to the C language. Cyclone [83] is a dialect of C that supports fat-pointer spatial safety and heap temporal safety. CCured [86, 87] is a software transformation system for C programs that similarly supports spatial and temporal safety. It also employs static analysis to identify pointer use that does not require runtime validation. However, these kinds of systems require significant changes to existing C source code, which limits their potential for adoption.

Sanitizers A sanitizer is another kind of system that can offer complete memory safety. Sanitizers maintain metadata and perform expensive instrumentation to detect and report a number of undesirable program behaviors at runtime. They are usually implemented in software but can benefit from hardware acceleration. The downside of these systems is their significant performance overhead; for this reason, they are often used as offline debugging tools rather than mitigations that can be enabled in production [106]. Address Sanitizer [104] and Valgrind [116] are perhaps the most widely known sanitizers.

2.1.3 Commodity hardware support

Relatively recently, various commodity architectures have added extensions that can be used to support memory safety mitigations. This trend indicates a willingness among commercial hardware designers and vendors to adopt changes that support improved security. To date, these extensions have not been successful in facilitating robust memory safety at a low performance cost; however, their adoption is encouraging for technologies like CHERI.

Intel's Memory Protection Extensions (MPX) [17] were developed to accelerate software-based bounds checking for spatial safety. A detailed analysis of MPX found that, while the technology is promising, it currently does not deliver the desired performance improvements and suffers from a number of issues related to its immaturity [89].

Arm has developed a number of extensions to facilitate security. One of these is Pointer Authentication Codes (PAC) [67, 103], which adds support for cryptographically signing pointers and validating that pointers have been signed before they are dereferenced. Using PAC to prevent pointer injection in this way is technically a probabilistic mitigation. Apple has adopted PAC on some of its platforms, but it has been successfully bypassed [10, 49, 97].

Another Arm security extension is Branch Target Identification (BTI) [74]. It adds a special no-operation instruction that is used to indicate valid indirect branch targets, and indirect branches to all other instructions result in a trap. BTI facilitates mitigating certain types of pointer injection rather than a class of memory safety vulnerability.

Extensions for tagging memory, such as Arm's Memory Tagging Extension (MTE) [47] and SPARC M7's Application Data Integrity (ADI) [91], have also been developed. These extensions provide hardware support for schemes similar to software lock-and-key instrumentation. They assign a tag, a few bits in length, to every pointer and every memory region of a particular size. Pointer tags are stored in unused upper address bits, and memory tags are stored in an auxiliary tag memory. When a pointer is dereferenced, its tag must match that of the referenced memory region, or the access will result in a trap. Because of the small number of possible tags, and the fact that pointer and memory tags can be modified by unprivileged code, these memory tagging extensions facilitate

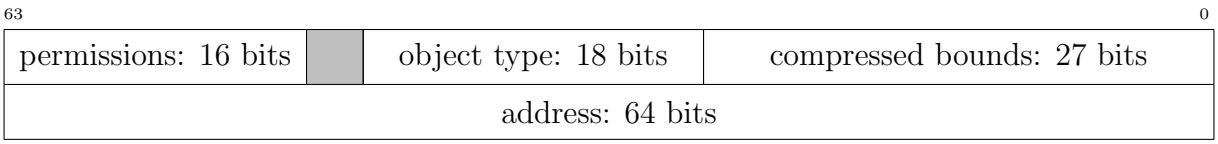


Figure 2.1: 128-bit CHERI capability format, adapted from CHERI ISAv8 [118]

only weak probabilistic protection. Considered alone, they are more useful for sanitizers or debugging aids rather than security mitigations.

2.2 The CHERI architecture

CHERI (Capability Hardware Enhanced RISC Instructions) [118, 120, 125] is an architectural extension for RISC ISAs that composes the capability-system model with current hardware and software stacks. In a capability system, resources are only accessed via capabilities, which are unforgeable tokens of authority that can be transferred between system components. Such a model makes it natural and efficient to implement two important security design principles: the principle of least privilege, which dictates that software should run with the minimum privilege necessary to perform its function; and the principle of intentional use, which dictates that software must explicitly select which privileges to exercise for each action it takes. These principles limit the scope and spread of damage caused by software flaws.

CHERI's security model hybridizes cleanly with contemporary RISC ISAs, CPU designs, MMU-based operating systems, and existing C/C++ software. It de-conflates memory virtualization and protection, allowing the MMU to handle virtualization while CHERI features provide protection. CHERI's security features can be adopted incrementally by software, and existing binaries can run unmodified on CHERI processors. CHERI also aims to have a low compatibility overhead: codebases that use C-style pointers in a standards-conforming way require minimal to no changes to re-compile with support for CHERI features.

CHERI introduces the architectural capability, a form of hardware-checked fat pointer that is used instead of an integer pointer to access memory. On 64-bit processors, architectural capabilities are 128-bit values that encode a virtual address, compressed bounds [124], permissions, and an object type. They are associated with an out-of-band one-bit tag that distinguishes valid from invalid capabilities. Figure 2.1 illustrates the 128-bit CHERI capability format. Architectural capabilities are implemented via microarchitectural additions to the CPU and SoC that support new capability instructions and registers as well as tagged memory for hardware-enforced capability integrity.

An important aspect of CHERI is that capabilities themselves represent the authority to access memory and can propagate freely throughout the system. No additional indirection or synchronous table lookups are required to authorize memory access. This design avoids adding significant microarchitectural performance overheads.

CHERI capabilities can be used to implement a number of powerful security primitives, including compartmentalization [120], but this dissertation is primarily focused on memory safety. The main properties of CHERI capabilities that software can use to facilitate robust memory safety are as follows:

Tags On a CHERI-enabled system, each 128-bit region of memory is tagged to indicate

whether it contains a valid capability or not. Attempting to access memory via an untagged capability results in an exception. Capability transformations by legitimate capability instructions preserve tags, but other modifications, such as corrupting capability bounds via a raw memory write, automatically clear the capability's tag, rendering it unusable. These properties guarantee architecturally that capabilities are unforgeable, and in particular valid pointers cannot be derived from ordinary data.

Bounds and permissions CHERI capabilities contain fine-grained bounds and permissions, which provide architectural guarantees that a capability cannot be used to access memory regions outside of its bounds or, for example, to write memory when it has only read permissions. Informally, the bounds of a CHERI capability can be described in terms of the capability's base, or inclusive lower bound, and its length, or number of bytes between the lower bound and exclusive upper bound.

Monotonicity All valid capability instructions monotonically decrease capability rights. In other words, bounds and permissions can never be expanded once they have been narrowed.

Sentry capabilities Sentry capabilities are immutable code capabilities that only allow control flow transfer [99]. Their target addresses cannot be modified, and they cannot be used to derive other capabilities. They are useful for implementing code pointers that should not be changed after they are initialized, such as global offset table entries and function return addresses. Sentry capabilities are identified by a specific value in their object type field. Software can *seal* a valid code capability to create a sentry, and the architecture can be configured so that function call and return instructions generate sentry capabilities for return addresses automatically [118].

2.3 Software use of CHERI

For CHERI to provide security benefits to application programs, software must take advantage of its architectural features, which support a wide range of potential uses. CHERI capabilities can represent abstract capabilities, i.e. non-forgeable tokens of authority, and be used as pointers to regions of virtual memory. They can facilitate fine-grained compartmentalization [120] as well as memory safety.

2.3.1 Spatial safety with CheriABI

Currently, the most sophisticated scheme for CHERI-supported memory safety is CheriABI. CheriABI is an execution environment for C/C++ applications that guarantees spatial safety [32, 99]. It is a feature of the CheriBSD operating system, which is a fork of FreeBSD that has been adapted to support CHERI features [25]. Its spatial safety guarantee stems from the facts that all memory accesses are made via valid capabilities and that capability bounds are shrunk to match the bounds of memory objects. These properties are enforced by the CHERI-LLVM compiler, which emits pure-capability code that only uses capability instructions to access memory, and other system components such as the kernel, the runtime linker, and C standard library, which work together to ensure that capability bounds are refined to match those of abstract memory objects. CheriABI also offers

attractive compatibility and performance overheads relative to similar past mitigations [32]. I further describe and analyze CheriABI in Section 3.3.

2.3.2 Pure-capability linkage

The loading and linking of pure-capability code is broadly similar to conventional linkage, with some important differences. Pure-capability code means that all pointers are implemented as CHERI capabilities: both those visible at the C and C++ language level, such as a pointer to some region of heap memory, and *sub-language-level pointers*, such as pointers in the global offset table (GOT) or return addresses saved on the stack. Sub-language-level pointers play a large role in linkage, and using capabilities to implement them means that some techniques from conventional linkage no longer work. However, pure-capability linkage also allows for improved security through greater application of the principle of least privilege.

Process initialization One immediate consequence of having a GOT full of capabilities, otherwise known as a captable, is that those capabilities must be initialized. Valid capabilities cannot be initialized statically; they must be derived via legitimate transformations from other valid capabilities. In other words, a binary file cannot contain capabilities that will automatically have valid tags when the binary is loaded. In pure-capability linkage, the system's runtime linker performs the required initialization of the captable.

Each entry in the application binary's captable initially holds a capability relocation, which tells the runtime linker the location and size of the global that the capability in that entry is supposed to reference. The runtime linker runs immediately after a process is created, before the application program. It has access to privileged capabilities that are present in a process' register file when it is created, and the runtime linker uses them to derive valid capabilities that correspond to the application binary's capability relocations. It overwrites the capability relocations with the valid capabilities. Before transferring control to the application, it also refines the bounds and permissions of the privileged capabilities that were initially present in the process' register file, such as the stack pointer capability and the program counter capability (PCC), to limit the privilege of the running application.

This capability refinement and captable initialization process is a is a crucial part of enforcing spatial safety in CheriABI. Note that it is not an automatic effect of running code on a CHERI-capable processor. It requires adaptation of both the compiler and the runtime linker, which work together to implement it. This process must take place for all pure-capability applications, even when they are position-dependent binaries and do not reference any external dynamic shared objects (DSOs).

Linkage ABI Pure-capability code also facilitates the principle of least privilege in the linkage ABI for accessing global values and functions. Here I describe the PCC-relative ABI, which is used by default in the CHERI-LLVM compiler, but other designs are possible [99]. In this ABI, the PCC has only read and execute permissions. Its bounds span the current DSO's text segment and captable. The captable is located at a statically known offset relative to the text segment, and the runtime linker populates it when it loads the DSO.

In the PCC-relative ABI, global variables are accessed by indexing into the current DSO's captable. Note that even globals within the same DSO must be accessed via the captable. In conventional linkage, such globals might be accessed directly via a known offset from the program counter. However, in pure-capability code, an analogous type of access would violate the principle of least privilege. It would require the PCC to have write permission and access to the DSO's data segment.

With the PCC-relative ABI, functions in the same DSO can be accessed via a jump instruction, and functions in a different DSO are called via the procedure linkage table (PLT). The captable entries for global function pointers, which are used by the PLT, contain sentry capabilities whose addresses reference the target function and whose bounds correspond to the target function's DSO.

The PCC-relative ABI enforces bounds for globals, which is important for spatial safety. It also limits the bounds of PCC to the current DSO, and restricts access to globals on a DSO level. This design results in a form of inter-DSO compartmentalization that protects the internal components of one DSO from being accessed by other DSOs. Even if an attacker managed to gain arbitrary code execution within some DSO, they would be able to access only code and data within the DSO and globals from outside of the DSO that have been exposed to it specifically. The attacker would be able to call functions outside of the DSO that have been linked. However, because the capabilities granting access to those functions are sentries, their entry points cannot be changed; the attacker could control only the arguments that the functions receive. Note that this form of compartmentalization also relies on the kernel to configure the architecture to implement return addresses as sentry capabilities. Otherwise, return address capabilities could be used to access private data in the DSO being returned to.

The isolation provided by the PCC-relative ABI means that control over machine state, including memory and code execution, in one DSO is not sufficient to compromise private information in another. This property is particularly appealing in the context of security-critical libraries used by potentially buggy application code. A memory safety bug in an application, for example, would not be sufficient to expose a key in some cryptography library or a memory allocator's privileged capability to the heap. However, the PCC-relative ABI does not fully enforce the principle of least privilege, because it uses a single captable for all functions in a DSO, and so each function has access to all globals in the DSO. The PLT ABI [99] restricts captables to the function level, facilitating greater isolation at the cost of complexity and performance.

2.4 Morello

CHERI initially extended the MIPS architecture, and later added support for RISC-V [118]. More recently, an extension to the ARMv8-A architecture known as Morello was developed in collaboration with Arm [69, 70]. It is a superset of ARMv8.2-A and implements the complete set of CHERI features with minor platform-specific adaptations.

Morello is the subject of the £187M UKRI Digital Security by Design program [113]. As part of this program, Arm produced an industrial-quality Morello demonstrator board, which was released in early 2022. The board, fabricated in a 7nm process, is based on the Arm Neoverse N1 System Development Platform [72]. It includes two CPU clusters, each with two superscalar out-of-order cores clocked at 2.5 GHz. Each core has a 64 KiB private L1 data cache, a 64 KiB private L1 instruction cache, and a 1 MiB private L2

unified cache. Each cluster has a 1 MiB shared L3 unified cache. The SoC's interconnect is clocked between 1.6 and 1.8 GHz. It has two DDR4 controllers, each of which supports an RDIMM up to DDR4-2667 speed (2667 MHz). The board can be configured to store capability tag bits in a carved-out memory region [56] or in RAM error-correcting-code bits.

Morello supports two instruction sets: A64 is the standard Armv8.2-A instruction set extended with instructions to interact with capabilities in a limited way, and C64 is a variant that supports a richer set of instructions for interacting with capabilities at the expense of instructions that use 64-bit integers to access memory [69]. Specifically, instructions are encoded in the same way across the two instruction sets, but instructions that use 64-bit operands in A64 use capability operands in C64, and vice versa. The active instruction set is determined by a bit in the processor state register (PSR). On a capability branch, this PSR bit is automatically saved into the least significant bit of the capability link register; its value is then updated to the least significant bit of the branch target's address. The PSR bit can also be toggled using the special `bx #4` instruction. This technique of using a branch target's least significant bit to switch between instruction sets has previously been used to support Arm's Thumb compressed instruction set [73].

2.4.1 Platform limitations

A preliminary but fully featured Morello adaptation of the complete CHERI software stack, including the CHERI-LLVM compiler and CheriBSD operating system, has been completed. However, Morello is the first high-fidelity superscalar capability machine ever to be built. Because of its novelty and limited project timeline, it currently has a number of known platform limitations. They include the following:

- There are some features of the Morello SoC microarchitecture that result in non-essential penalties for pure-capability code. These penalties affect capability jumps, loads, and stores, and they inflate the overhead of pure-capability compilation. Attempts to fully diagnose and work around the penalties, including a benchmarking ABI that sacrifices some of CheriABI's security properties to simulate more realistic performance, are in progress. These non-essential penalties are expected to be eliminated in future microarchitectures as techniques for building high-performance CHERI-enabled processors are better understood. They will be documented in a future technical report, which has not yet been made public at the time of writing.

- Capability-related optimizations present for other CHERI architectures have not been added to the Morello backend of the CHERI-LLVM compiler.

- Specialized Arm assembly routines for byte-string operations like `memcpy()` and `memset()` in the CheriBSD standard library have not been ported to work with capabilities, so pure-capability binaries on Morello use slower C routines.

- The thread-local storage specification for Morello is still undergoing changes, so pure-capability binaries use emulated thread-local storage.

Additionally, current understanding of the general performance effects of capability use in superscalar capability machines is relatively limited.

These limitations mean that it is currently challenging to measure performance differences between normal 64-bit Arm code and pure-capability code. Similar challenges exist

for measuring the performance effects of adding new pure-capability code paths, because the added code may be affected differently by current microarchitectural penalties than the existing body of code.

Despite these challenges, the work in this dissertation targets Morello because it will ultimately determine the feasibility of widespread CHERI adoption. I designed my experiments and hypotheses to be robust in the face of microarchitectural and toolchain-related changes that will take place as general understanding improves and the above limitations are addressed. Chapter 5 contains the first performance measurements from the Morello board to be reported.

2.5 Virtual-machine language interpreters

Higher-level languages like JavaScript, Java, and C# play a large role in modern computing infrastructure. They power mobile applications, cross-platform client and server programs, and the user-facing dynamic components of websites. Unlike lower-level languages, which are typically compiled directly to machine code for a target architecture, higher-level languages are often executed by a program known as an interpreter.

Interpreters themselves are generally written in lower-level languages and compiled for a specific architecture. They must meet a significant challenge: to facilitate safe and high-performance execution of the interpreted language. In other words, an interpreter must efficiently implement the interpreted language's high-level features while also protecting its own code and data structures from interference by arbitrary interpreted code.

While an interpreter might implement spatial safety and a temporally-safe garbage-collected heap for the language being interpreted, the lower-level language that implements the interpreter, e.g. C or C++, likely lacks such security properties. In this case, the interpreter's implementation can contain memory-safety vulnerabilities that allow interpreted code to interfere with the interpreter's internal data structures. These bugs can be difficult to detect by inspecting the source. Features such as just-in-time compilation, discussed further below, which improve the performance of interpreters but greatly increase their implementation complexity, make it easy to introduce subtle bugs.

In the presence of such memory-safety vulnerabilities, a carefully crafted input program can leverage them to turn arbitrary *constrained* execution in the high-level interpreted language into arbitrary *unconstrained* code execution in the interpreter's process. This scenario is particularly dangerous in the context of web browsers, which we use regularly to download and interpret JavaScript code from untrusted, potentially malicious sources. Mitigating memory-safety vulnerabilities in language interpreters is thus crucially important for security.

A common design pattern for modern interpreters is to make use of a virtual machine [75]. A virtual-machine language interpreter implements the interface of an abstract computer, typically one that is stack-based or register-based. It operates on a set of values that occupy its virtual register file and memory regions. To interpret an input program, it first compiles the program into bytecode that specifies virtual machine operations. Each bytecode instruction in program order is then dispatched to a function in the interpreter's implementation that performs the operation and modifies the virtual machine's state accordingly.

2.5.1 Just-in-time compilation

Most modern language interpreters support some form of just-in-time (JIT) compilation to improve performance [6]. With JIT compilation, the interpreter compiles frequently interpreted parts of a program, such as functions or particular loops, to native machine code that is executed directly. One way that JIT-compiled code can reduce execution time is by eliminating overhead related to the dispatch of virtual machine bytecode instructions.

JIT compilers can also reduce execution time by using type speculation to facilitate traditional compiler optimizations. Interpreted languages are often dynamically typed, which means that a bytecode operation has to check the types of its operands at runtime to determine the appropriate course of action. This dynamic typing prevents the application of techniques from traditional optimizing compilers, which rely on static type information to simplify generated code. By speculating, or making assumptions about, types and values for bytecode operands, JIT compilers can apply traditional optimization techniques to improve performance significantly.

While JIT compilation facilitates faster execution, it takes longer to JIT-compile and start executing a piece of code than it does to start interpreting it directly. In practice, many interpreted code paths are only executed once, so the cost of JIT-compiling them outweighs the benefit of faster execution relative to interpretation. To take advantage of the fast startup time of interpretation as well as the fast execution of JIT compilation, most modern interpreters feature a hierarchy of execution tiers supported by different JIT compilers.

All code is initially interpreted directly. As a code path is executed more and more, or becomes *hotter*, it is JIT-compiled by increasingly aggressive compilers that perform more speculation and sophisticated optimizations. Generally, each JIT compiler in the hierarchy takes longer to compile code but produces faster output. It is common for lower, less-optimizing tiers to collect type and value profiling information that is used for speculation by the higher tiers [128].

JIT compilers' aggressive speculations can be wrong, so they also insert lightweight checks to verify them. When optimized JIT code detects that one of its assumptions was incorrect, it transfers execution to a less-optimized execution tier that can handle the case appropriately. Usually, execution tiers share the same register and calling conventions so that they can also share the same stack. In this case, transferring execution between them is known as on-stack replacement (OSR), which is relatively inexpensive.

2.6 JavaScriptCore

JavaScriptCore is the JavaScript interpreter in Apple's WebKit browser framework, which is used to build the Safari browser and a number of other popular applications [54]. It compiles JavaScript into bytecode, interprets that bytecode with a register-based virtual machine, and supports JIT compilation. It is primarily written in C++. It has the following high-level characteristics:

Execution JSC features four tiers of JavaScript execution. These tiers, in increasing order of optimization, are known as the Low Level Interpreter (LLInt), the Baseline JIT, the Data Flow Graph (DFG) JIT, and the Faster Than Light (FTL) JIT. All JavaScript code is initially compiled to bytecode that gets interpreted by the LLInt, and, as it becomes hotter, it is compiled and run in the higher execution

tiers. The Baseline JIT removes the overhead of dispatching bytecode operations but does not perform other optimizations. The DFG and FTL JITs use profiling information collected by the LLInt and Baseline JIT to speculate on the types and values of operands to bytecode instructions; they perform increasingly aggressive optimizations.

All execution tiers share the same stack and use on-stack replacement to switch tiers as necessary. The shared calling and register-use conventions among the JavaScript execution tiers differ from those of usual C/C++ ABIs. As a result, when the C++ body of the JSC application wants to call into the LLInt or some JIT-compiled code to execute JavaScript, it first calls a trampoline function that sets up the stack according to the JavaScript execution tiers' conventions.

JavaScript values JSC bytecode operates on 64-bit values that can represent double-precision floating point numbers, 48-bit pointers to memory objects on the heap, 32-bit integers, and other special values. Each value, known as a `JSValue`, has a tag that bytecode instruction implementations can use to identify its type. The `JSValue` format uses a technique called NaN-boxing, which makes use of the 2^{51} unused bit patterns in the Not-a-Number space of the IEEE754 floating point specification, to encode all of the possible values in 64 bits [121]. Pointers are encoded with no modification, doubles are shifted by 2^{49}, and 32-bit integers contain a tag in the upper 16-bits of the 64-bit word.

The JavaScript heap The heap in JavaScriptCore is a collection of 16KB slabs called blocks, each of which is divided into equal-sized cells. The cell size of a block depends on the block's size class. All memory objects on the heap inherit from the C++ class `JSCell`, which is an eight-byte header that contains metadata about the object's type and garbage collection status. Heap memory objects are used to represent all kinds of JavaScript constructs, such as strings, arbitrarily sized integers, and JavaScript objects. When a `JSValue` encodes a pointer to the heap, it points to the `JSCell` for some memory object.

JSC's heap has a conservative, mark-and-sweep, non-moving garbage collector [52]. It supports generational collection with sticky mark bits [33], and it uses write barriers to perform incremental and concurrent collection.

JavaScript objects JavaScript objects are collections of properties that can be accessed by name [40]. For example, the property `height` of object `person` could be accessed by the expression `person["height"]`. Arrays are special objects that contain properties known as elements whose names are 32-bit integers, as well as a `length` property whose value is equal to the name of the highest element plus one.

In JSC, JavaScript objects are represented by the `JSObject` class and its subclasses. Object properties are `JSValue`s. Besides the metadata inherited from `JSCell`, `JSObject`s contain a statically configurable number of inline property slots, and a pointer to a dynamically allocated region that stores additional properties and array elements.

Mapping between a JavaScript object's property names and storage locations is performed by the `Structure` class. `JSCell` metadata contains a Structure ID, which indexes a global table to identify the appropriate `Structure` for an object.

The `Structure` contains a hash table that maps property names to their storage location. To avoid repeating costly operations, the LLInt and JIT execution tiers cache property name locations for particular structure IDs inline with either the bytecode or JITed code for operations that perform property lookups [92].

2.7 Operating-system virtual memory

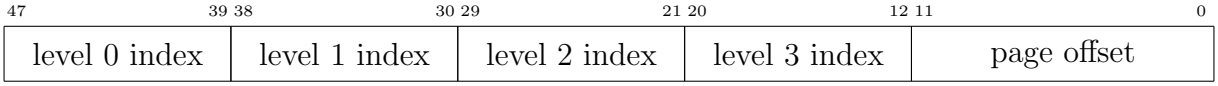


Figure 2.2: Schematic diagram of how a 48-bit input virtual address is typically used to index into a four-level page table with 4KB pages in Armv8.2-A.

Virtual memory subsystems perform a number of critical functions in contemporary general-purpose operating systems. Among other things, they allow resources to be shared safely between processes, protect memory by enforcing access permissions, and liberate applications from having to consider the details of a machine's physical memory layout. On a system with virtual memory, each application process is isolated in its own virtual address space. Regions of these virtual address spaces, usually known as pages, are mapped to corresponding regions of physical memory. The mappings need not be one-to-one, so multiple virtual pages may be backed by the same physical page. Contiguous pages in the virtual space need not be contiguous in physical memory. The mappings also have permissions, so that restricting read, write, or execute access can be performed at a coarse granularity.

Virtual memory features are implemented by a combination of kernel support and a hardware component called the memory management unit (MMU). The design of MMUs varies across architectures. Here I describe the operation of the MMU specified in Armv8.2-A [68]. The MMU design for Morello is based on Armv8.2-A, with some novel features added to support sweeping capability revocation. I introduce these novel features in Section 5.2.

The Armv8.2-A MMU is typically configured to translate 48-bit input virtual addresses into 48-bit output physical addresses. It uses a four-level page table data structure with a 4KB page size. As illustrated in Figure 2.2, 9-bit segments of the input virtual address index into successive levels of the page table to determine the corresponding page table entry (PTE). PTEs are 64 bits in length. They contain a 36-bit field that identifies the physical address of the target page. The lower 12 bits of the input virtual address are identical to the lower 12 bits of the output physical address; they are combined with the 36-bit page address to make a 48-bit output physical address that has the appropriate offset into the target physical page.

Mappings between input virtual and output physical addresses are cached in a Translation Lookaside Buffer (TLB) to speed up translation. An input virtual address is first looked up in the TLB, and if the correct mapping is present there, then translation can proceed without traversing the page table. Otherwise, the hardware iterates through the page table structure to determine the appropriate output physical address. If the traversal succeeds, then the mapping is added to the TLB, potentially evicting other mappings. The architecture also includes instructions for manually evicting entries from a TLB, which

software must use in some cases for correctness. For example, when software updates a PTE to reduce its permissions, it must evict mappings that hold stale versions of the entry.

Page table entries also contain a number of configuration bits that control address translation and memory access. The bits of Armv8.2-A PTEs that are relevant to later discussions about sweeping capability revocation are as follows:

Access Permissions The `AP[2:1]` bits control read and write access permissions for the mapping. A value of 01 allows read and write access, a value of 11 allows read only access, and a value of X0 does not allow read or write access. Attempting to perform a disallowed access results in a fault.

Execute Never The `XN` bit controls execute access permissions for the mapping. A value of 0 allows execute access, and a value of 1 does not allow execute access. Attempting to perform a disallowed access results in a fault.

Access Flag The `AF` bit indicates whether a mapping has been accessed since the last time `AF` was set to 0. A value of 1 indicates that an access has taken place. The architecture can be configured so that `AF` is managed in hardware or software. If `AF` is managed in software, then an attempt to load a page table entry with an `AF` value of 0 into the TLB results in a fault. Software is expected to set `AF` to 1 so that the TLB load can proceed. If `AF` is managed in hardware, then on an attempt to load a page table entry with an `AF` value of 0, the hardware performs an atomic read-modify-write on the entry to update `AF` to 1 before it is loaded into the TLB. Software can set `AF` to 0 and later check its value to determine whether mappings were accessed in the interim.

Dirty Bit Modifier Dirty tracking refers to techniques for identifying page table mappings that have experienced writes. It is possible to perform dirty tracking in software by setting access permission bits to disallow writes and installing a write-fault handler that changes the permissions to allow writes. Then, mappings that have experienced writes are those that have their permissions set to allow writes.

On some architectures, it is possible to perform dirty tracking in hardware, so that a distinct PTE bit tracks whether a mapping has experienced a write since the last time the bit was cleared. Armv8.2-A supports hardware dirty tracking, and the `DBM` bit indicates whether hardware dirty tracking is enabled. If `DBM` is 0, then the bit has no effect. If `DBM` is 1, and `AP` is 11, then on an attempt to write via the mapping, the hardware performs an atomic read-modify-write on the entry to set `AP[2]` to 0 and the write proceeds as normal, without a fault.

When the `DBM` bit is 1, it changes the meaning of some `AP` bit patterns. A PTE with `DBM` 1 and `AP` 11 allows read and write access but is considered clean, meaning it has not experienced a write to trigger hardware dirty bit tracking. A PTE with `DBM` 1 and `AP` 10 allows read and write access and is considered dirty, meaning it has experienced a write to trigger hardware dirty bit tracking. `AP[2]` thus indicates a mapping is dirty when it is 0 and indicates a mapping is clean when it is 1. A PTE with `DBM` 0 and `AP` 11 allows read-only access, and writes result in a fault.

Software can mark PTEs as clean and later check the value of `AP[2]` to determine whether dirtying writes occurred via the mapping in the interim. In order for hardware dirty bit tracking to be enabled, hardware management of `AF` must also

be enabled. `AF` and `AP[2]` can be modified in a single atomic read-modify-write operation.

2.8 Garbage collectors

Garbage collectors are systems that facilitate programming environments with automatic memory management. They present an alternative to manual memory management, which requires a programmer to notify an allocator explicitly when regions of memory are no longer in use. In a garbage collected system, the programmer explicitly allocates memory, but the garbage collector automatically determines when regions of memory are no longer in use and reclaims them. Garbage collectors were originally introduced to simplify memory management in high-level interpreted languages like Lisp [79], and many garbage collection techniques involve instrumenting memory operations, which only works with the direct cooperation of an interpreter or runtime environment. However, certain types of garbage collector can be used in non-cooperative environments, e.g. to manage the heap in typical C-language runtimes [14].

Garbage collectors necessarily come with performance costs. Many designs introduce additional work related to traversing or sweeping memory and also increase the amount of memory used relative to manual management. Designs that instrument loads or stores can result in significant runtime overheads. Another performance impact of some garbage collector designs is the introduction of *stop-the-world phases*, or *pause times*, during which the application cannot run. A wealth of garbage collector designs has been developed for different execution environments and performance trade-offs. Here I discuss some of the most prominent ones.

Reference counting Reference counting is a form of garbage collection that involves maintaining a count for each memory object that stores the number of references to it. When a reference to a memory object is created, the object's count increases, and when a reference is destroyed, the count decreases. When the count reaches zero, the memory object is no longer reachable and the system reclaims it as garbage.

The primary benefits of reference counting are its simplicity and the fact that garbage can be collected as soon as it is generated without pausing the application to traverse or sweep memory. However, reference counting requires instrumenting all loads and stores that might create or destroy references, which can come with a significant performance overhead. Reference counting is also susceptible to memory leaks when groups of memory objects that reference each other cyclically become unreachable from the stack or registers.

Mark-and-sweep A mark-and-sweep system performs garbage collection passes periodically, e.g. when some allocation threshold is reached. Passes consist of two stages that take place while the application is paused. In the mark stage, the garbage collector transitively traverses every reachable memory object, starting with a *root set*, which is typically the stack and register file. The collector follows each pointer in the root set. If the corresponding object is unmarked, the collector marks it then follows any pointer that the object contains. It repeats the process until all reachable objects have been marked. In the sweep stage, the collector scans all of memory. When it encounters a marked object, it clears the mark for the next collection. When it encounters an unmarked object, it collects the object's memory and adds it to a free list.

The benefits of mark-and-sweep collection are that it does not require instrumentation of any loads or stores and it does not move memory objects during a collection. Its drawbacks are that it scans all of memory, and it does not compact memory objects, which can cause fragmentation, increase the complexity of allocating free regions, and reduce locality. Furthermore, the naïve approach described here requires the application to be paused during the entire collection, which has a significant performance impact.

Copying Copying garbage collection, also known as moving garbage collection, splits memory into two subspaces: the from-space and the to-space. Memory is allocated in the to-space, and a garbage collection pass typically takes place when the to-space runs out of free memory. A pass first pauses the application then swaps subspace names: the to-space becomes the from-space and vice versa. The collector then traverses live objects transitively starting from the root set, similarly to a mark-and-sweep collector. But rather than marking them, the collector copies them into the new to-space, compacting objects into a single contiguous region. Specifically, when the collector finds a live object, it copies it into the next available part of the to-space and replaces the pointer referencing the object with a pointer to the object's new location in the to-space. It also replaces the old version of the object in the from-space with a *forwarding pointer* to its new location in the to-space, indicating that it has already been copied, so other references to the object can simply use the forwarding pointer and not erroneously copy it again. After the pass, garbage is left behind in the from-space, which is ready to be used as the to-space in a future pass. The application is then unpaused, and memory can be allocated in a bump-the-pointer fashion from the free region of the current to-space.

The benefits of copying collection are that it does not require instrumentation of any loads or stores; it compacts memory objects, which can reduce fragmentation, make allocating free memory trivial, and improve locality; and it does not have to scan all of memory. The drawbacks are that it requires double the memory of other techniques, it performs extra work to copy memory objects, and the naïve version described here requires the application to be paused during the entire collection.

Conservative or precise Garbage collectors can be either conservative or precise in how they identify pointers. Conservative garbage collectors do not identify pointers exactly [14]. They scan every word of memory in the root set and all live objects, and if a word could be interpreted as pointer to some memory object, then it is treated as such. The benefits of conservative collectors are that they do not require any tagging or instrumentation to identify pointers. The drawbacks are that they can result in false positives and negatives. A false positive occurs when a non-pointer word is interpreted as a pointer; in this case, a region of memory that is actually garbage might not be collected. False negatives can occur in runtime environments that do not cooperate with the garbage collector. If pointers are obfuscated and reconstructed, e.g. in a linked list in C that stores next and previous pointers in a single word by performing an exclusive-or operation on them, then the garbage collector might not detect them. False negatives can result in dangling pointers to memory objects that were erroneously collected as garbage, which is a serious security flaw. However, runtime environment cooperation can eliminate the possibility of false negatives.

Precise garbage collectors identify pointers exactly. The benefits of precise collection are that it does not suffer from false positives or negatives. The drawbacks are that it requires either instrumentation or tagging to track which words are pointers. These

schemes generally come with performance or implementation overhead. However, hardware features like CHERI that support tagged pointers allow conservative collectors to be made precise with no additional overhead.

Generational Generational garbage collection is an optimization for mark-and-sweep and copying garbage collectors. It takes advantage of the observation that most allocated memory objects have short lifespans. In other words, recently allocated objects are likely to become garbage. In abstract, a generational collector classifies memory objects as young if they have been allocated since the most recent garbage collection pass and old if they have survived a previous garbage collection pass. It supports two kinds of garbage collection passes: a lightweight pass that traverses young objects but skips old objects, and a full pass that traverses all objects.

Mark-and-sweep collectors can use *sticky mark bits* [33] to facilitate generational collection, and copying collectors can use additional subspaces to support different generations [65]. One important aspect of generational garbage collectors is that they must instrument stores that place a reference to a young object into an old object. Old objects with references to young objects must be traversed in lightweight passes, and their references marked, so that the referenced young objects won't erroneously be collected and leave dangling pointers in the old objects.

Incremental and concurrent The above descriptions of mark-and-sweep and copying collectors assumed that garbage collection passes take place while the application is paused. However, this assumption results in unacceptably high application pause times in practice. Incremental and concurrent garbage collection are optimizations that aim to reduce the amount of consecutive time that the application spends paused during a collection pass.

Incremental collection [7] describes techniques that allow a garbage collection pass to be completed in multiple stages, between which the application is allowed to run. Concurrent collection [13, 37, 38, 112] describes techniques that allow phases of the garbage collection pass to take place concurrently with the running application. Incremental and concurrent collectors may require the instrumentation of pointer loads or stores for correctness, so that application activity in the middle of a garbage collection pass cannot cause memory objects to be collected erroneously. Concurrent collection requires additional synchronization in the form of atomic read-modify-write operations and instructions to order memory operations.

Chapter 3

The memory-operations framework

In this chapter, I introduce the *memory-operations framework* (MOF), a conceptual framework and structuring principle for memory-safety analysis. The MOF can be applied in a number of ways: to characterize memory-safety mitigations in terms of the assumptions they make and the memory protections they offer, to characterize memory-safety attacks in terms of the memory-safety vulnerabilities and attack techniques that they exploit, and to determine whether a given mitigation would protect against a given attack.

The MOF has two main aspects that distinguish it from past approaches. First, it applies trusted computing bases and threat models, which are well-known concepts in security analysis more generally, to memory-safety mitigations, which appears not to have been done in a systematic way by past conceptual frameworks. Second, it characterizes memory-safety mitigations and attacks in terms of the memory-safety vulnerabilities and intermediate memory operations that they mitigate or exploit respectively. Memory operations in particular represent a novel way of thinking that facilitates more clear and useful analysis than past approaches.

After defining the framework, I apply it to characterize the protections offered by CheriABI. Past work on CheriABI's security benefits has been primarily empirical, focused on CheriABI's observed ability to mitigate known-vulnerable example programs. I use the MOF to perform a novel, systematic analysis of CheriABI. This analysis includes an expanded threat model for CheriABI that has not been considered before.

I then apply the framework to analyze exploits recently carried out in the wild against devices running Apple's iOS to determine how CheriABI memory-safety protections would fare against contemporary real-world attacks. I determine that CheriABI is effective as a mitigation, but I also identify a number of important gaps in its memory-safety protections. This exercise demonstrates the utility of both CheriABI and the MOF. The discovered gaps motivate the work described in the remaining chapters of this dissertation.

3.1 Framework components

The MOF consists of three components that must be defined for a memory-safety mitigation: a trusted computing base, a threat model, and a set of memory protections. In this section, I describe them and how they are useful in reasoning about what kinds of attacks a memory-safety mitigation protects against. The framework's design was inspired by prior work on both CHERI-supported memory-safety mitigations and on memory-safety analysis more generally. However, I leave discussion of prior work to the following section in order

to better highlight the differentiating features of the MOF.

3.1.1 Trusted computing base

A trusted computing base (TCB) [5, 115] is a set of components that a system relies on to deliver its security properties. In particular, TCB components are assumed to be free of vulnerabilities. TCBs are important in reasoning about what kinds of attacks a memory-safety mitigation protects against because they provide an explicit scope for which system components are protected by the mitigation and which components must be independently correct to support the mitigation's proper functioning.

For example, consider the stack canary mitigation described in Section 2.1.1. The compiler is part of its TCB because the mitigation relies on the compiler to emit code that stores suitably randomized stack canaries in stack frames and checks their value when a function returns. While stack canaries may normally protect against stack-based buffer overflow attacks with high probability, bugs in the compiler might allow an attacker to bypass this protection. A compiler vulnerability, for instance, that allows the attacker to leak the values of randomized stack canaries would make it possible for them to bypass the mitigation and exploit stack-based buffer overflows.

3.1.2 Threat model

A threat model is a set of assumptions about what level of privilege a potential attacker has in a system. Like the TCB, it also provides scope for what kind of attacks a memory-safety mitigation protects against. As a simplified, schematic example, consider traditional file access control on UNIX-like systems, which might be employed as a mitigation to prevent attackers from reading a secret file. A reasonable threat model for this mitigation might assume that an attacker has compromised a non-privileged user account on the system. Attacks are then only considered in scope for being mitigated if they are carried out by a compromised non-privileged user account. If the attacker successfully reads the secret file by, for example, compromising an administrator account, then the attack violates the assumptions of the mitigation's threat model and is not in scope.

3.1.3 Memory protections

Memory protections represent a way to characterize mitigations in terms of the kinds of memory-safety attacks they prevent. More specifically, memory protections enumerate the vulnerabilities and intermediate memory operations that a memory-safety attack could possibly exploit. Mitigations can then be classified according to which vulnerabilities and intermediate memory operations they render impossible. Particular attacks can also be classified in terms of the vulnerabilities and intermediate memory operations that they rely on.

Recall from Section 2.1 that, for a memory-safety attack to occur, there must be an attacker-accessible memory-safety bug, i.e. a memory-safety vulnerability. Memory-safety vulnerabilities can be classified as spatial-safety, temporal-safety, or type-safety vulnerabilities.

With access to a memory-safety vulnerability, an attacker is able to perform memory reads or writes that violate a memory object's integrity, i.e. its bounds, lifetime or type. An attacker can often link together such reads and writes to violate the integrity of additional

memory objects and ultimately to obtain arbitrary code execution or some other form of privilege escalation. I refer to these reads and writes, carried out by an attacker as intermediate steps in a memory-safety attack, as *memory operations*. There are eight kinds of memory operation, comprising two kinds of operation on four kinds of operand: attackers can leverage memory-safety vulnerabilities to read (exfiltrate) or write (inject) code, data, code pointers, or data pointers. For ease of expression, I also refer to the intermediate memory operations carried out as part of a memory-safety attack as *attack mechanisms*.

Both memory-safety vulnerabilities and attack mechanisms are associated with a particular storage class, e.g. stack objects, heap objects, or globals. For example, in the buffer overflow attack outlined in Section 2.1.1, the memory-safety vulnerability is the `strcpy()` invocation that allows the bounds of the buffer to be violated, and it is a spatial-safety vulnerability on the stack. The stack-based attack mechanisms used to exploit the vulnerability are code pointer injection, used to overwrite the return address on the stack, and code injection, used to write arbitrary code into the buffer that is ultimately executed.

A memory-safety mitigation's set of memory protections comprises the vulnerability classes, attack mechanisms, and specific attack techniques that it renders impossible. For some examples, consider the memory protections of some of the past mitigations introduced in Section 2.1.2. SoftBound aims to mitigate all spatial safety vulnerabilities, rendering them unexploitable across all storage classes. W⊕X mitigates the attack mechanism of code injection across all storage classes by guaranteeing that injected code is non-executable. Stack canaries probabilistically mitigate only a specific kind of attack technique: overwriting a function's return address on the stack via a contiguous buffer overwrite. They do not mitigate, for example, targeted return address overwrites that skip over the canary.

Enumerating a mitigation's set of memory protections is important because doing so makes it straightforward to determine, subject to scoping by the TCB and threat model, whether a mitigation prevents an attack. In particular, a mitigation prevents an attack if it mitigates any of the vulnerability classes, attack mechanisms, or specific techniques that the attack employs.

3.1.4 Discussion

Recall that an unmitigated vulnerability has to be present for an attacker to carry out any kind of memory-safety attack. A mitigation is thus most powerful if it eliminates entire classes of memory-safety vulnerability, rendering all bugs in those classes unexploitable. Preventing attack mechanisms is the next most powerful kind of protection, and it is an important form of defense in depth. Mitigations that target a specific attack technique without considering vulnerability classes or attack mechanisms are better than no protection at all, but they are limited in scope.

Probabilistic or secret-based protection is similarly less powerful than deterministic protection. For example, mitigations like ASLR (Section 2.1.1) do not deterministically eliminate any vulnerability class, attack mechanism, or attack technique. They make it less likely in theory for an attacker to successfully carry out certain attack techniques; however, the threat model might allow for the attacker to access one of many well known side-channels that leak the randomization.

For memory-safety mitigations, certain kinds of security properties are often explicitly out of scope. Application logic bugs, such as an omitted access control check, cannot directly be addressed in terms of memory-safety. Similarly, denial of service is usually out of scope, because when a mitigation prevents a memory-safety attack, the only reasonable course of action is often to terminate the running program, even with suitable signal or exception handlers in place.

3.2 Prior work

Prior work that inspired the design of the MOF can be divided into models for reasoning about memory safety or exploitability in general and security analyses of systems built with CHERI. In this section, I describe both categories.

3.2.1 Frameworks for memory-safety analysis

MITRE CWE classes The MITRE Corporation's Common Weakness Enumeration (CWE) database [29] is an expansive list of software and hardware vulnerability classes, abstracted from past real-world attacks, that can be used to analyze memory-safety attacks and mitigations. It is useful for organizing past attacks that have been carried out against real software artefacts, but it also has a number of shortcomings, particularly related to specificity and coverage of the CWE classes for memory safety [88]. It inspired the creation of an alternative framework with a sole focus on memory safety that considers all possible properties of memory-safety attacks rather than abstracting from empirically observed attacks.

Eternal war in memory In *Eternal war in memory*, Szekeres *et al.* [111] present a model for classifying memory-safety attacks and mitigations that inspired the memory protection component of the memory-operations framework. Relative to their model, the MOF more completely and succinctly enumerates the possible kinds of memory operations, and it draws a clearer distinction between memory operations and vulnerability classes. It also takes storage classes for vulnerabilities and memory operations into account, and it includes type safety as a distinct form of memory-safety vulnerability rather than treating it as a synonym for memory safety as a whole. Consideration of TCBs and threat models is also a differentiating factor.

Weird machines Thomas Dullien's work on weird machines and exploitability [39] helped to develop ideas in the MOF about how attackers exploit memory-safety vulnerabilities to achieve privilege escalation. In particular, it helped to formulate the idea that memory-safety attacks can be considered as a series of memory operations or attack mechanisms that are linked together.

3.2.2 CHERI security analysis

Prior work that has analyzed the security of memory safety supported by CHERI includes the paper by Davis *et al.* that introduced CheriABI [32], a report on CHERI by the Microsoft Security Response Center (MSRC) [57], and a report on the viability of open-source CHERI desktop software by Capabilities Limited [119]. Here I consider how each performed a security analysis and highlight limitations relative to the MOF.

CheriABI Davis *et al.* note that CheriABI uses CHERI architectural features to protect both language-level and sub-language-level pointers, and they show how the enforcement of capability bounds guarantees spatial safety. The architectural and software components that make up the TCB are well described, but the paper does not contain meaningful discussion of a threat model or assumed attacker access. It evaluates security by considering how well CheriABI mitigates the specific vulnerabilities in BOdiagsuite, a set of potted C programs that all contain some form of buffer overflow. However, it does not consider attack mechanisms or discuss vulnerability classes beyond spatial safety.

MSRC The MSRC report evaluates security properties in terms of the three vulnerability classes: spatial safety, temporal safety, and type safety. However, it does not consider attack mechanisms in a systematic way. It also does not have a clear concept of TCBs: in places, it seems to conflate CHERI architectural features, pure-capability code, and the CheriABI process environment.

Capabilities Limited The Capabilities Limited report considered the feasibility and value of adapting a large corpus of desktop software to support pure capability compilation. It performed analyses for the CheriABI process environment and included whiteboarding exercises involving support for fine-grained software compartmentalization with CHERI. CheriABI and its TCB were well described in the report. The report also had a sophisticated treatment of threat models, and considered the specific threat model for each desktop software component separately. Interestingly, the authors had to reverse engineer these threat models from past vendor patches and threat advisories, since they were almost never documented explicitly. In terms of memory protection, however, the document considered a vulnerability to be mitigated only if CHERI memory safety renders its entire vulnerability class unexploitable, and stated that it was "not currently clear how to best reason about" the value of eliminating particular attack mechanisms.

3.3 CheriABI analysis

In this section, I apply the MOF to analyze CheriABI. More specifically, I consider CheriABI in terms of its TCB, threat model, and memory protections to determine what attacks are in-scope and mitigated. This is the first analysis of CheriABI's memory protections that considers attack mechanisms in addition to vulnerabilities, and it also includes an expanded threat model that has not been proposed before.

3.3.1 TCB

The TCB for CheriABI comprises the CHERI architecture [118] and the following software components:

The CHERI-LLVM compiler for C and C++ emits *pure-capability* code, i.e. code that only uses capability instructions to access memory. This pure-capability code sets appropriate bounds on capabilities that are used to access stack-allocated memory objects. Specifically, to access a stack-allocated memory object, a capability is derived from the register file's *stack pointer capability* such that its base points to the base

of the memory object and its bounds are reduced to the memory object's size. The access then proceeds via this newly derived capability. The compiler also supports using capabilities to enforce *sub-object bounds* between fields of a single `struct` [99]. In addition to emitting pure-capability code, the CHERI-LLVM compiler and LLD static linker also emit *capability relocations* that contain information about the location and size of global memory objects. The runtime linker uses these relocations to initialize bounded capabilities to globals. On Morello, pure-capability code uses the C64 instruction set.

The CheriBSD kernel initializes the userspace process by placing privileged capabilities into its initial register file. It sets the *default data capability* (DDC), which governs memory access by non-capability integers, to null, which means that only capabilities can be used to access memory. It configures system registers so that sentry capabilities are automatically used by instructions that perform branch-and-link and return operations. The kernel's system calls expect capability arguments and validate them to ensure that it does not violate spatial safety in the process while servicing system calls. Signal delivery similarly works with valid capabilities. The kernel is also responsible for managing process virtual address spaces.

The standard library heap allocator bounds capabilities to dynamic memory allocations before returning them to application code. This guarantees that a capability to some heap-allocated memory object cannot be used to access memory outside of its bounds.

The runtime linker refines privileged capabilities provided by the kernel, initializes bounded capabilities to globals, and facilitates the loading of dynamic shared objects (DSOs). Its operation is described in detail in Section 2.3.2.

3.3.2 Threat model

A basic threat model for CheriABI could assume that the attacker initially controls an unprivileged, untrustworthy input stream to some non-malicious but potentially buggy program. For example, the attacker might control the input file to an image processing tool, or the contents of a web page that is rendered by a browser. Other examples might consider an attacker with control over one process that sends input to another process via IPC, or to the kernel via system calls. In all of these scenarios, the attacker's aim is to exploit bugs accessible to them via some input-handling code to increase their privilege on the system, by e.g. gaining arbitrary code execution or exfiltrating sensitive data. All such attacks are potentially in scope for CheriABI. However, attacks that involve attacker control over some privileged or trusted input stream, like a debugging interface, are out of scope.

It is also possible to formulate a second, expanded threat model for CheriABI that can be used in conjunction with the basic one to widen its scope. This expanded model has not been proposed by any previous work on analyzing the security properties of CheriABI. It considers an attacker that has arbitrary code execution within an unprivileged DSO of some process and aims to gain code execution or access to secrets in a more privileged DSO. On a conventional system, if an attacker gains arbitrary code execution in some application, then they control the entire process. However, as described in Section 2.3.2, if an attacker gains arbitrary code execution in a CheriABI process, then they may only

spatial safety	temporal safety	type safety
✓	✗	✗

(a) Vulnerability classes

	code	data	code pointer	data pointer
read	✗	✗	✗	✗
write	✓	✗	✓	✓

(b) Attack mechanisms

Table 3.1: Vulnerability classes and attack mechanisms mitigated by CheriABI memory protections. Ticks indicate that a vulnerability class or attack mechanism is mitigated, while crosses indicate a lack of protection.

control a single DSO. If the controlled DSO does not contain implementations of TCB components, such as the heap allocator or the runtime linker, then code running in that DSO cannot access TCB internals. It can only access particular global functions to which it has been linked; it can fill arguments with arbitrary data and capabilities that are available within the controlled DSO. With this expanded threat model, such inter-DSO attacks are potentially in scope for CheriABI. Attacks that involve attacker control of a DSO that implements part of the TCB, or that involve attacker access to system calls like `exec()` that allow control over the entire process environment, are out of scope.

3.3.3 Memory protections

As described in Section 2.3.1, CheriABI eliminates spatial safety vulnerabilities for all storage classes by representing all pointers as capabilities to finely bound memory objects. Sub-object bounds for structure members are not enabled by default, but can be enabled by passing a flag to the compiler [99]. CheriABI as originally published does not mitigate temporal-safety or type-safety vulnerabilities.

But even in the presence of unmitigated memory-safety vulnerabilities, CheriABI's use of CHERI architectural features eliminates a number of attack mechanisms. W⊕X on capability permissions make it impossible for an attacker to inject executable code, and capability tags make it impossible for an attacker to inject code and data pointers. However, while CheriABI makes it impossible to inject pointers, which is a significant limitation, it does not limit an attacker's ability to read extant valid capabilities, modify them via tag-preserving but privilege-decreasing capability instructions, and move them to other locations. I use the term *valid capability misappropriation* to describe this behavior. Note that capabilities pointing to data are more malleable than code pointers implemented as sentry capabilities.

Table 3.1 summarizes CheriABI's memory protections in terms of vulnerability classes and attack mechanisms. These protections do not mitigate all memory-safety attacks: temporal-safety or type-safety vulnerabilities can still be used in conjunction with the unmitigated attack mechanisms. For example, consider a temporal safety vulnerability that causes two memory objects on the heap to alias. An attacker might leverage existing code paths to read code, data, code pointers, and data pointers in the aliased memory.

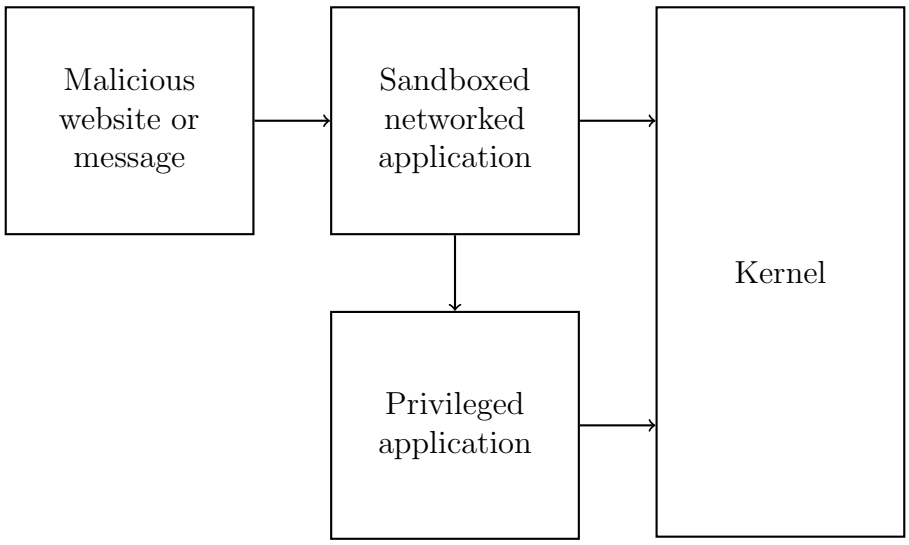


Figure 3.1: Schematic diagram of system components that an attacker gains control of as they progress through an exploit chain.

They might also write arbitrary data to the aliased memory to influence the program's behavior and escalate their privilege.

In addition to memory-safety attacks that CheriABI's memory protections do not mitigate, application logic vulnerabilities and denial-of-service attacks are also out of scope. However, assuming that its TCB is implemented correctly, and taking into account scope limitations from the TCB and threat model definitions, CheriABI's memory protections make significant strides in limiting possible memory-safety attacks.

3.4 Project Zero iOS exploit chain analysis

In this section, I analyze a series of real-world attacks that were carried out against devices running Apple's iOS. They were discovered in the wild by Google's Project Zero team [9]. Using the MOF, I determine the vulnerability classes and attack mechanisms used and consider whether CheriABI in its current form would have mitigated the attacks. Based on this analysis, I identify gaps in CheriABI protections and motivate future work on memory safety with CHERI. These results demonstrate the effectiveness of the MOF as well as CheriABI.

To elevate their privilege and ultimately take control of today's complex and hardened computer systems, adversaries must link together multiple attacks, or exploits, against system components in an *exploit chain*. Figure 3.1 illustrates a schematic example of the components targeted in an exploit chain against a contemporary client system, such as a laptop or smartphone. In this example, a remote adversary with no foothold on the system ultimately ends up with the ability to read and write kernel memory arbitrarily, i.e. complete control over the system.

The attacker initially initially exploits a vulnerability in some networked application, such as a web browser or messaging client, to obtain arbitrary code execution in the process running the application. They might accomplish this by tricking the victim into visiting a website they control or by sending the victim a specially formatted malicious message. The networked application might be sandboxed, meaning that it has limited

access to kernel programming interfaces and can only communicate with a particular whitelist of other processes. To bypass these limitations, the attacker gains arbitrary code execution in a more privileged process on the whitelist that is subject to fewer restrictions. They perform this bypass by exploiting a vulnerability either in the operating system's inter-process communication (IPC) libraries, which are used by all processes that perform IPC, or in IPC message handling code specific to the privileged process. Finally, the adversary exploits vulnerabilities in some accessible kernel programming interface to gain arbitrary kernel memory read and write. With control over kernel memory, they own the system. They can execute arbitrary code in supervisor mode, disable platform protections such as sandboxing and code signing, and inject surveillance implants.

Note that the above scenario describes attacking a consumer device that is not a server. It is perhaps more straightforward for an attacker to gain an initial foothold on a server because they can connect to networked server applications directly.

In late 2019, Google's Project Zero published detailed descriptions of five distinct exploit chains that were employed by hackers between 2016 and 2019 to target iPhones running iOS 10.0.1 to 12.1.2 [9]. All of these exploit chains involve a remote attacker inducing a victim to visit a controlled malicious website that exploits vulnerabilities in Safari's JavaScriptCore (JSC) JavaScript interpreter. They then leverage control over the JSC process to attack other system components and ultimately gain arbitrary control over kernel memory.

For each part of each exploit chain, I analyze the memory-safety-related aspects of the attack. I identify the initial vulnerability exploited, provide a summary of the attack, and describe whether and how current CheriABI protections would mitigate it. I also consider the vulnerability classes and attack mechanisms exploited in intermediate steps of the attack. More detail about each chain can be found on the Project Zero website [9].

The Project Zero report mentions seven JSC exploits that were used as initial links in the five exploit chains, depending on the version of iOS targeted. The JSC exploits are covered in significantly less detail than the other parts of the chains, and in some cases even the corresponding Common Vulnerability Enumeration entries and bug reports do not provide enough information to determine relevant vulnerability classes and attack mechanisms. Specifically, three of the exploits are described in enough detail to analyze directly. The other four involve just-in-time (JIT) compiler side effect modeling. Side effect modeling bugs are technically logic bugs in the JIT compiler, but they are often exploited to violate type safety or spatial safety [48]. Because the descriptions of these JIT side effect modeling attacks are not detailed enough to facilitate useful analysis, I also consider a separate Project Zero report that covers a single exploit of a JIT side effect modeling bug in detail [49].

3.4.1 Exploit chains

Chain 1 kernel exploit

Vulnerability: Omitted bounds check in the sandbox-accessible kernel driver function `AGXAllocationList2::initWithSharedResourceList()` allows legitimate kernel data pointers to be injected out of bounds on the kernel heap.

How CheriABI protects: Mitigates spatial safety vulnerability.

Summary: First performs heap grooming: calling kernel functions to make many heap allocations to control the heap layout. Uses the vulnerability to inject a legiti-

mate data pointer out of bounds. Leverages the injected data pointer to cause heap aliasing and a kernel ASLR leak. Uses data, data pointer, and code pointer injection to fabricate a kernel object that grants the attacker-controlled process the ability to read and write kernel memory (known as task for pid 0, or tfp0).
Vulnerability classes: spatial safety, then temporal safety
Attack mechanisms: legitimate pointer injection, data injection, data pointer leak, data pointer injection, code pointer injection

Chain 2 kernel exploit

Vulnerability: Reference management error when the sandbox-accessible kernel driver function `IOSurfaceRootUserClient::s_set_surface_notify()` is called via `IOConnectCallAsyncMethod()`
How CheriABI protects: Prevents data pointer injection.
Summary: Grooms the heap then uses the vulnerability to create a dangling pointer that causes kernel heap aliasing. Uses injected data and data pointers to break kernel ASLR with brute force. Injects data and pointers to fabricate a kernel object that grants tfp0.
Vulnerability classes: temporal safety
Attack mechanisms: data pointer leak, data injection, data pointer injection

Chain 3 sandbox escape

Vulnerability: Incorrect bounds check in the `libXPC` IPC library allows one process to cause an out-of-bounds read of a code pointer in any other process.
How CheriABI protects: Mitigates spatial safety vulnerability.
Summary: Targets the `mediaserverd` daemon. Uses a separate `mediaserverd`-specific bug that leaks unrecognized IPC messages to spray its heap with many copies of a code-reuse-attack payload. Uses the IPC vulnerability to trigger the payload, which constructs a fake object that grants access to vulnerable kernel objects (`AppleVXD393` or `D5500` clients) and exfiltrates it to the sandboxed process. Note that this attack does not need to break userspace ASLR, because the randomization is performed once at boot time and shared across all processes.
Vulnerability classes: spatial safety
Attack mechanisms: code pointer injection, data pointer injection, data injection

Chain 3 kernel exploit

Vulnerability: Calling `CreateDecoder()` followed by `DestroyDecoder()` in the `AppleVXD393` or `D5500` drivers causes a kernel structure to be freed with a dangling pointer left behind that can be freed again.
How CheriABI protects: Prevents data pointer injection.
Summary: Uses heap grooming and the temporal safety vulnerability to repeatedly cause controlled memory aliasing on the kernel heap. Uses this controlled aliasing and data pointer injection to create a kernel heap state that matches the initial conditions of the chain 2 kernel exploit. Employs the same techniques as the chain 2 kernel exploit.
Vulnerability classes: temporal safety

Attack mechanisms: data pointer injection, data pointer leak, data injection

Chain 4 sandbox escape

Vulnerability: Incorrect use of libXPC reference semantics in the `cfprefsd` daemon allows memory objects on its heap to be freed twice.
How CheriABI protects: Prevents data pointer injection.
Summary: Uses IPC messages to spray the heap of the `cfprefsd` process with a ROP payload and separately with pointers containing a likely address for the payload. Exploits the double-free vulnerability to alias the payload address with a heap memory object that causes the payload to be executed. The payload constructs a fake object that grants access to a vulnerable kernel object (`ProvInfoIOKitUserClient`) and exfiltrates it to the sandboxed process. Note that this attack does not need to break userspace ASLR, because the randomization is performed once at boot time and shared across all processes.
Vulnerability classes: temporal safety
Attack mechanisms: data pointer injection, code pointer injection, data injection

Chain 4 kernel exploit

Vulnerability: Omitted bounds check in `ProvInfoIOKitUserClient::ucEncryptSUInfo()` allows the caller to control the length of an unbounded call to `memmove()`.
How CheriABI protects: Mitigates spatial safety vulnerability.
Summary: Grooms the heap and uses the unbounded `memmove()` and data pointer injection to cause memory aliasing and match the initial conditions of the chain 2 kernel exploit. Employs the same techniques as the chain 2 kernel exploit.
Vulnerability classes: spatial safety, then temporal safety
Attack mechanisms: data pointer injection, data pointer leak, data injection

Chain 5 kernel exploit

Vulnerability: Reference management error present in the sandbox-accessible `task_swap_mach_voucher()` function.
How CheriABI protects: Prevents data pointer injection.
Summary: Grooms the kernel heap and uses the vulnerability to create a dangling pointer that aliases attacker-controlled data with a voucher object on the kernel heap. Injects data and calls `thread_get_mach_voucher()`, which causes the kernel to allocate another object on the heap that aliases with a region of attacker-controlled memory and doubly-link it to the attacker-controlled voucher using legitimate pointers. Uses a data pointer injection in the aliased voucher to create a kernel heap state that matches the initial conditions of the chain 2 kernel exploit. Employs the same techniques as the chain 2 kernel exploit.
Vulnerability classes: temporal safety
Attack mechanisms: data injection, data pointer leak, data pointer injection

JSC exploit 1

Vulnerability: Reference management error in `BindingNode::bindValue()` allows

out-of-bounds JSC heap write.
How CheriABI protects: Mitigates spatial safety vulnerability.
Summary: Overwrites the structure ID of a controlled JavaScript object to create a fake TypedArray, which facilitates the construction of a memory read and write primitive. Uses this primitive to inject code. Carries out a JOP attack to call the privileged JIT `memcpy()` function, which overwrites a previously compiled function with the injected code. Calls the previously compiled function to execute the code.
Vulnerability classes: spatial safety
Attack mechanisms: code pointer injection

JSC exploit 2

Vulnerability: Uninitialized memory copy in `JSArray::appendMemcpy()` allows controlled memory to be treated as the content of a JavaScript array object.
How CheriABI protects: Prevents data pointer injection.
Summary: Grooms the heap to control the uninitialized data. Exploits the vulnerability to construct the addrof and fakeobj primitives [102] that grant memory read and write. Uses the primitives to inject code as in exploit 1.
Vulnerability classes: temporal safety-related
Attack mechanisms: data pointer injection, code pointer injection

JSC exploit 6

Vulnerability: Missing case in garbage collector leads to exploitable use-after-free.
How CheriABI protects: Mitigates spatial safety vulnerability.
Summary: Exploits the bug to gain access to an internal field of a JavaScript object. Corrupts the field to trigger an out-of-bounds memory access, which is used to construct a limited read and write primitive. Uses the limited primitive to disable heap hardening measures added to JSC and proceed as in exploit 2.
Vulnerability classes: temporal safety, then spatial safety
Attack mechanisms: data pointer injection, code pointer injection

JSC JIT

Vulnerability: Just-in-time compiler side-effect modeling bug allows for out of bounds array access.
How CheriABI protects: Mitigates spatial safety vulnerability.
Summary: JIT compiles some code that takes advantage of a bug in the JIT optimization pass for common-subexpression elimination to access a JavaScript array out of bounds. This out-of-bounds access is sufficient to construct a limited read and write primitive, which can be used to disable mitigations, including the use of Arm's PAC, and proceed with the addrof and fakeobj primitives as in previous exploits.
Vulnerability classes: logic, then spatial safety
Attack mechanisms: data pointer injection, code pointer injection

Exploit	Initial vulnerability class	Intermediate attack mechanism
C1 Kernel	Spatial	—
C2 Kernel	Temporal	Data pointer injection
C3 Sandbox	Spatial	—
C3 Kernel	Temporal	Data pointer injection
C4 Sandbox	Temporal	Data pointer injection
C4 Kernel	Spatial	—
C5 Kernel	Temporal	Data pointer injection
JSC 1	Spatial	—
JSC 2	Temporal	Data pointer injection
JSC 6	Temporal, then spatial	—
JSC JIT	Logic, then spatial	—

Table 3.2: A summary of how CheriABI would protect against the Project Zero iOS exploit chains. Features of the exploits that would be mitigated by CheriABI protections (spatial vulnerabilities and data pointer injection) are presented in green, and unmitigated features in red. CheriABI would mitigate all of the examined attacks, either by rendering the initial vulnerability unexploitable or preventing an intermediate attack mechanism.

3.4.2 Results

Table 3.2 summarizes the how CheriABI protections would fare against the Project Zero iOS exploit chains. For each part of each chain, it considers whether CheriABI eliminates the vulnerability class of the initial vulnerability; if not, it considers whether CheriABI prevents any of the intermediate attack mechanisms. In the cases of JSC 6 and JSC JIT, CheriABI does not render the initial vulnerability unexploitable, but a secondary vulnerability that CheriABI does mitigate is used immediately afterwards. Every analyzed attack would have been mitigated if current CheriABI protections were applied to the targeted programs, which is encouraging about CheriABI's value in protecting against contemporary attacks. However, the results also highlight gaps in current CheriABI protections that should be addressed:

Temporal safety While CheriABI protections would have mitigated all of the analyzed attacks, they render the initial vulnerability unexploitable for only four out of eleven. The majority of the remaining attacks would have been mitigated by the prevention of pointer injection, which suggests that pointer injection, and data pointer injection specifically, is a crucial intermediate step for contemporary memory-safety attacks. In fact, even if CheriABI did not provide spatial safety, it would still have stopped all eleven attacks at a pointer injection step. However, because pointer injection is an attack mechanism, mitigating it does not render the seven non-spatial vulnerabilities in the table unexploitable in general. Without pointer injection in their toolbox, attackers would likely find other ways to exploit the vulnerabilities, such as data-oriented programming. Mitigating additional vulnerability classes is thus an important gap in current CheriABI protections, and all but one of the seven non-spatial vulnerabilities in the table are related to heap temporal safety. Chapter 5 of this dissertation describes the design, implementation, and evaluation of a scheme for providing temporal safety for userspace heaps with CHERI.

System components These exploit chains also identify key system components that are targeted in contemporary attacks. One such component is the web browser's JavaScript interpreter, which allows remote attackers to obtain an initial foothold even on client systems. Modern JavaScript interpreters are extremely complex codebases that combine interpretation engines, just-in-time compilers, and garbage-collected heaps to deliver constrained execution of arbitrary code from untrustworthy sources. Their complexity makes them a plentiful source of exploitable vulnerabilities, and their remote accessibility makes them a common target for attackers, as demonstrated by the Project Zero exploit chains. To date, a detailed investigation of how language interpreters might be adapted to benefit from CheriABI-like protections has not been published. Chapter 4 of this dissertation describes the process of adding support for CHERI features to JavaScriptCore and evaluates the feasibility, source-code compatibility, and security implications of doing so.

Another key system component is the kernel. To date, the CheriBSD kernel has enforced CheriABI protections for userspace processes as part of the TCB. Ongoing work led by my colleague Alfredo Mazzinghi [78] is adding support for pure-capability compilation and spatial safety for the kernel itself.

Attacks against CheriABI The chain 5 kernel exploit suggests what successful attacks against CheriABI, even with the above gaps addressed, might look like. In the exploit, the attacker uses a temporal safety vulnerability to alias a structure on the kernel heap with some memory whose contents they control. With this controlled memory, the attacker modifies data fields of the structure and then makes a system call that causes the kernel to allocate another heap structure and doubly link the two structures together using legitimately derived kernel data pointers.

In the actual exploit, the attacker proceeds by injecting data pointers to spoof kernel memory objects, which CheriABI memory protections would mitigate. However, the attacker's ability to obtain legitimately derived kernel data pointers suggests other kinds of attacks that might be possible. If the attacker exploited the temporal safety vulnerability a second time to also control the memory backing the doubly-linked structure allocated by the kernel, then they would have a significant base from which to spoof kernel memory objects, even without pointer injection. They could modify the data backing both structures; inject legitimate kernel capabilities accessible to them; strategically overwrite capabilities with data; and directly inject null pointers, which are represented by untagged values. It is not unreasonable to expect that these primitives allow for the construction of a kernel memory object that can be used to escalate privilege.

Such an attack could be effective against CheriABI, even with current work on userspace temporal safety and adaptations for language interpreters and the kernel incorporated. This further highlights the need to mitigate vulnerability classes in addition to particular attack mechanisms. Temporal safety for the kernel heap remains an open problem.

3.5 Discussion

In this chapter, I built on past work to develop the novel memory-operations framework for reasoning about memory-safety mitigations and what kinds of attacks they prevent. I applied the MOF to CheriABI to characterize its current protections in detail. Then, using the framework and my analysis, I considered whether CheriABI would have mitigated a number of contemporary real-world exploits against systems running iOS. I determined that, while current CheriABI protections are significant and would have mitigated all of the examined exploits, there are important gaps that should be addressed. These gaps motivate my work on adapting JavaScriptCore to support CheriABI-level protections, presented in Chapter 4, and on userspace heap temporal safety with sweeping capability revocation, presented in Chapter 5.

Chapter 4

JavaScriptCore on CHERI

CheriABI has previously been shown to guarantee spatial safety for "typical" C and C++ applications when they are adapted to support pure-capability compilation [32]. However, language interpreters tend to differ from typical applications in a number of ways. They have the unusual goal of allowing arbitrary, but constrained, code execution for potentially untrustworthy input programs. They often implement custom memory allocators and garbage collectors, and interpreters with JIT compilers must be able to compile, link, and modify code in their own address spaces while they are running.

Language interpreters play a foundational role in modern computing infrastructure, so improving their security properties is important. No previous work has attempted to adapt a contemporary language interpreter that supports JIT compilation to CHERI. In this chapter, I perform a study on adapting JavaScriptCore (JSC), including support for the Baseline JIT execution tier, to pure-capability code. My primary goal is to determine how the complexity and non-typical behaviors of a contemporary language interpreter affect the difficulty of the adaptation and the security properties of the result.

I aim to address the following questions in this chapter:

- Is adapting a language interpreter with JIT compilation to pure-capability code feasible? How does JSC's design affect its amenability to a pure-capability adaptation, and do these features generalize to other interpreters? Does the resulting program work correctly?
- How does the source-code compatibility overhead of a pure-capability JSC adaptation compare to that of other software artefacts?
- Is JSC more secure as a result of pure-capability compilation? Are its security properties on par with those of a typical CheriABI binary? Does the use of CHERI affect existing security mitigations in JSC?

I determine that it is feasible and indeed practical to adapt a language interpreter with a JIT compiler to pure-capability code without affecting correctness. The source-code compatibility overhead of doing so is on par with that of similar types of software. Unlike typical pure-capability applications, however, adapting a language interpreter to pure-capability code does not inherently guarantee complete spatial safety or other CheriABI security properties. However, straightforward changes can bring its security properties up to par.

This is a pilot study that aims to provide a baseline from which further research can be conducted. In particular, performance measurement is left as future work for a number of

reasons. Primarily, current limitations of the Morello platform, as described in Section 2.4.1, make it impractical to isolate the performance cost of switching from traditional 64-bit Arm code to pure-capability code. Other CHERI platforms were not suitable for performance measurements, or even for a JSC adaptation, because JSC does not have native interpreter or JIT support for MIPS or RISC-V. Even if JSC did support those architectures, the performance cost of a pure-capability adaptation is likely to be dominated by the increase in pointer, and thus working-set, size. As a result, properties of microarchitectural features such as the cache hierarchy and memory subsystem will be critically important in measuring performance costs. There are currently no MIPS or RISC-V capability machines with suitable high-fidelity contemporary microarchitectures. Similarly, Morello architectural simulation would not adequately capture important microarchitectural details, and there is not a microarchitectural simulator for Morello. The goal of this work is thus to facilitate eventual performance evaluation using the Morello board once current platform limitations have been addressed.

4.1 Feasibility

This section considers the feasibility of adapting language interpreters with JIT support to pure-capability code, using JSC as a model. I first summarize the major changes required for pure-capability support in JSC. I then highlight particular aspects of JSC that affected how amenable it was to adaptation and discuss how they generalize to other language interpreters. Finally, I evaluate whether the pure-capability adaptation of JSC has feature parity with the normal 64-bit Arm version.

4.1.1 Summary of changes

The major changes required to adapt JSC fall into one of three categories: JavaScript execution, `JSValue` representation, and C++ pointer use. Some additional background information about JSC's JavaScript execution environment is required for the below discussion. Recall from Section 2.6 that JSC has four tiers of execution: the Low Level Interpreter (LLInt), the Baseline JIT, the DFG JIT, and the FTL JIT. The pure-capability adaptation features support for the LLInt and Baseline JIT; the other JIT tiers are left as future work.

The LLInt is implemented in Offline Assembly, which is a portable assembly language unique to JSC. Offline Assembly's abstract instructions and registers can be compiled to native instructions and registers for multiple assembly languages, including the Arm and Intel instruction sets. There is also an option to compile Offline Assembly to C++, so that the LLint can work on architectures that do not have a specific Offline Assembly backend, but the C++ version of the interpreter is slower and not compatible with the JIT execution tiers.

The Offline Assembly compiler is written in Ruby and is part of the JSC source tree. One major benefit of using a portable assembly language to implement the LLInt is that it allows JSC to use custom calling and register conventions for JavaScript execution. Using C or C++ to implement the LLInt, for example, would force JavaScript execution to use a C or C++ ABI, which may be inefficient and unnecessary for JavaScript execution.

The Baseline JIT is implemented as a collection of C++ routines that emit buffers of native assembly instructions. Recall that all of JSC's execution tiers share the same

stack, so interpreted and JIT-compiled JavaScript code interoperates without any need for translation. All JavaScript code, whether interpreted or JIT-compiled, can also call into a collection of C++ helper functions that make up the JSC runtime.

The pure-capability adaptation of JSC supports three configurations for execution:

LLInt C++ backend This configuration compiles the LLInt using the C++ Offline Assembly backend. When compiled this way, the LLInt is not compatible with other execution tiers.

LLInt Morello backend This configuration compiles the LLInt using the Morello Offline Assembly backend, but it does not enable the baseline JIT. All JavaScript code is interpreted.

LLInt Morello backend with Baseline JIT This configuration compiles the LLInt using the Morello backend and also enables the Baseline JIT. JavaScript code starts out interpreted by the LLInt, but as it becomes hotter, it is JIT-compiled by the Baseline JIT.

I now describe the changes required to support these configurations.

```
macro op_is_boolean()
    get(m_operand, t1)
    loadConstantOrVariable(t1, t0)
    xorq ValueFalse, t0
    tqz t0, ~1, t0
    orq ValueFalse, t0
    return(t0)
end
```

Figure 4.1: A representation of the Low Level Interpreter implementation, in Offline Assembly, of the JSC bytecode instruction `is_boolean`.

Low Level Interpreter The LLInt is a collection of macros written in Offline Assembly that implement the functionality of JSC bytecode operations. A cosmetically simplified implementation of the bytecode instruction `is_boolean` is presented in Figure 4.1. It first loads the bytecode operand's location into register `t1` then loads the operand, which is a `JSValue`, into `t0`. It then performs an exclusive or of the operand with `ValueFalse`, the `JSValue` constant for false. If any bits besides the least significant bit remain set, then the operand was not `ValueFalse` or `ValueTrue`, the `JSValue` constant for true, and so `t0` is cleared. Otherwise, `t0` is set to the value 1. The implementation then performs an or of `t0` with `ValueFalse`, so that the result will be the `JSValue` representing false if the bytecode operand was not a boolean, and true if it was.

Instructions in Offline Assembly have suffixes that indicate the widths of their expected operands. For example, `addi` is used to perform addition with 32-bit integers, `addq` is used for 64-bit integers, and `addp` is used for pointers. The different suffixes are used during Offline Assembly compilation to select appropriate native instructions and register widths. Note that the instructions in `is_boolean` all have `q` suffixes.

As a whole, the LLInt source code contained many instructions with `q` suffixes that actually operate on pointers, or `JSValue`s that may hold pointers. This is not a problem for conventional 64-bit machines, where Offline Assembly instructions with `q` and `p` suffixes generally lower to the same native assembly code. On Morello, however, Offline Assembly instructions with `q` suffixes lower to native instructions that use 64-bit register views, and Offline Assembly instructions with `p` suffixes lower to native instructions that use capability register views. A read from a 64-bit view of a capability register returns its address, while a write to such a view modifies the address, invalidates the capability, and clears the upper bits of the register. As a result, instructions with `q` suffixes that actually operated on capability pointers had to be changed to have `p` suffixes.

An alternative solution to this problem would be to make Offline Assembly instructions with both `q` and `p` suffixes emit native instructions with capability operands. However, doing so would result in inefficient code generation, because not all operations that can be performed on 64-bit operands in a single native instruction can be performed on the addresses of capability operands in a single native instruction.

Interestingly, LLInt routines that operate on `JSValue`s that will never be dereferenced as a pointer can use `q`-suffixed Offline Assembly instructions without any problem. Those instructions invalidate and clear the higher bits of the capability register containing the `JSValue`, but the `JSValue`'s NaN-boxed representation stored in the capability register's address remains intact.

Note that the implementation of `is_boolean` uses instructions with `q` suffixes to operate on `JSValue`s. However, because the `JSValue` operands are not ever used as capability pointers, `is_boolean` does not require any modification to work correctly on Morello.

The majority of changes to the LLInt implementation involved modifying Offline Assembly instruction suffixes to match their operands. Another type of change involved replacing hard-coded literals with compile-time constants. Specifically, the LLInt source already contained compile-time constants that were correctly defined for the size of a machine-native pointer, the size of a machine-native register, and the size of a virtual machine register (i.e. a `JSValue`). However, for many operands in the LLInt implementation, these sizes were hard-coded to eight bytes using immediate values. The logarithms of these sizes, used for bit shifting and other operations, were also hard-coded. The set of constants was expanded to include their logarithms, and hard-coded values were replaced with the appropriate constants.

Offline Assembly C++ backend Modest changes were required to support capabilities in the Offline Assembly backend that emits C++ code. An argument was added to the backend invocation to indicate that a CHERI architecture is being targeted, so that the existing backend could be mostly reused with changes for the CHERI case. These changes were limited to code-emitting functions, such as `cloopEmitOperation()` and `cloopGenerateConditionExpression()`, in code paths for operations on pointers. For example, C code emitted for pointer operations was modified not to cast integer literal operands to `uintptr_t` because doing so could confuse the CHERI-LLVM compiler about capability provenance. It was also modified to insert compiler builtins such as `__builtin_cheri_address_get()` where appropriate.

Offline Assembly Morello backend Unlike the C++ backend case, which involved augmenting an existing Offline Assembly backend to support the CHERI C++ compiler, a

new backend was added to support Morello assembly. The Morello assembly backend used the standard 64-bit Arm assembly backend as a starting point and involved relatively minor changes. Primarily, the implementation of pointer-suffixed Offline Assembly instructions was modified to emit native capability instructions and to raise errors for unsupported pointer operations.

```
void JIT::emit_op_is_boolean(const Instruction* currentInstruction)
{
    auto bytecode = currentInstruction->as<OpIsBoolean>();
    int dst = bytecode.m_dst.offset();
    int value = bytecode.m_operand.offset();

    emitGetVirtualRegister(value, regT0);
    xor64(TrustedImm32(JSValue::ValueFalse), regT0);
    test64(Zero, regT0, TrustedImm32(static_cast<int32_t>(~1)),
           regT0);
    boxBoolean(regT0, JSValueRegs { regT0 });
    emitPutVirtualRegister(dst);
}
```

Figure 4.2: The Baseline JIT implementation of the JSC bytecode instruction `is_boolean`.

Baseline JIT compiler When a JavaScript function is compiled by the Baseline JIT, the compiler examines each JSC bytecode operation in program order and emits corresponding native machine instruction templates into a buffer. The buffer is then linked with the rest of the running JavaScript code. Baseline JIT compilation thus reduces interpretation overhead related to dispatching bytecode operations, but it does not perform any other optimizations.

The architecture of the Baseline JIT compiler is broadly similar to that of the LLInt: the Baseline JIT's C++ routines use abstract operations that are emitted as native instructions by a backend assembler for each target architecture. These abstract operations resemble Offline Assembly instructions and have suffixes like `32`, `64`, and `Ptr` to indicate the kinds of operands they expect and the register widths used by the instructions they emit.

Figure 4.2 presents the Baseline JIT implementation of the `is_boolean` bytecode instruction. It performs exactly the same operations as the LLInt version, except the locations of the bytecode's operand and destination registers are known at compile time, so do not have to be loaded dynamically. Though the `xor64` and `test64` functions indicate that they operate on 64-bit values, the `TrustedImm32` constructors indicate that the immediates can fit in 32 bits to facilitate potentially simpler generated code. The `boxBoolean` function emits an instruction that adds `ValueFalse` to its operand register.

Many of the changes required for the Baseline JIT were analogous to those required for the LLInt. Bytecode instruction implementations were modified to use appropriately sized abstract operations, and hard-coded sizes for pointers, values, and registers were replaced with appropriate compile-time constants.

However, other changes were unique to the Baseline JIT because it emits and links machine code. The 64-bit Arm assembler backend had to be updated to use newly

introduced encodings for capability instructions on Morello. The disassembler was similarly updated with new encodings to support debugging. Additionally, when linking JIT-ed code, the assembler backend would previously emit code that materialized pointers to functions from integer literal operands. While this is not a problem on conventional 64-bit Arm architectures, capabilities cannot be materialized from integers on CHERI architectures by design. To work around this issue, the backend was modified to emit valid capability function pointers inline with the code. These capabilities are then loaded and used via an offset from the PCC.

This change required disabling branch compaction, an optimization in which eligible basic blocks are moved around in memory to reduce the number of branches. Because instruction streams are normally 4-byte aligned, this movement might violate the 16-byte alignment requirement for inline capabilities. Branch compaction could be restored in future work by emitting capabilities as out-of-line constant pools. Another minor change required for linkage was that code pointers to C64 code on Morello have their least significant bit set; this must be cleared before working with code addresses.

JavaScript value representation The size of `JSValue`s was increased to 128 bits so that they could hold capability pointers to memory objects on the heap. NaN-boxing is performed as before, but using the held (valid or invalid) capability's 64-bit address to store NaN-boxed values. Interestingly, this turned out to be a low-friction change in terms of source compatibility. Because pointers are not modified in JSC's NaN-boxing scheme, i.e. their NaN-boxed representation is identical to their unboxed representation, when a `JSValue` is determined to contain a capability pointer to the heap, it can be dereferenced directly as a valid capability.

In the JavaScriptCore C++ codebase, `JSValue`s were represented as a union, with one of the views as a `uint64_t` and another as a pointer type. With pure-capability compilation, the pointer view automatically caused the size of `JSValue` to increase to 128 bits, and code that uses the pointer view worked without modification. The `uint64_t` view was used by functions that evaluate the NaN-boxed value to determine the type of the `JSValue`, such as `JSValue::isInt32()` and `JSValue::IsDouble()`. After a single-line modification to change the view's type from `uint64_t` to `uintptr_t` the existing functions worked as desired for the new `JSValue` representation: code emitted by the CHERI-LLVM compiler, e.g. to perform bitwise tests on the `uintptr_t` view, works with the capability's address automatically. A single-line change in each function that casts the `uintptr_t` union view back to a `uint64_t` could be added for clarity, as was done in the actual adaptation.

C++ pointer use JavaScriptCore's C++ codebase needed a number of modifications to account for the change in the size and semantics for capability pointers. In many situations, `uint64_t`-equivalent types were used where `uintptr_t` should be, or vice versa. This does not cause problems on normal 64-bit architectures, where these types have the same width. On CHERI architectures, however, `uint64_t` is not large enough to hold capability pointers, and using `uintptr_t` where it is not required wastes space and may cause inefficient code generation. Inline assembly blocks with pointer operands also had to have their constraints updated to use capability registers.

Many changes were also related to bitwise operations on pointers. For example, optimizations to pack pointers that reference aligned memory objects had to be disabled.

Code that used bitwise operations to store data in the upper bits of pointers had to use an alternative location, as doing so would invalidate capabilities. On the other hand, code that used bitwise operations to modify the lower bits of pointers was adapted to use particular routines and compiler builtins that guarantee the operations will not unintentionally invalidate underlying capabilities.

4.1.2 Generality

Here I highlight the features of JSC that affected how amenable it was to pure-capability adaptation and comment on how they generalize to other contemporary language interpreters.

Custom macro-assembly language Adapting the LLInt involved a significant amount of incidental complexity because it is written in Offline Assembly. In order to get it working as pure-capability code, the Offline Assembly compiler backends had to be adapted as well. However, the inherent complexity of actually adapting the LLInt was relatively low: most changes involved instruction operand widths and erroneously hard-coded sizes. Other contemporary JavaScript interpreters, such as Google's V8, also have interpreting execution tiers written in custom macro-assembly languages, and adapting them is likely to be similar [43].

Non-optimizing JIT compiler The inherent complexity of adapting the Baseline JIT was essentially the same as that of adapting the LLInt, and the same kinds of changes were required. Other modern interpreters, including V8, also feature a non-optimizing JIT compiler similar to the Baseline JIT that would likely be similar to adapt.

Various other modifications were required to support JIT compilation. Because the JIT compiler also has an assembler and disassembler, new Morello instruction encodings had to be added. This required a significant amount of effort, but was mostly mechanical. Additionally, capability literals had to be emitted inline with JIT-compiled code to support linkage because pointers could no longer be materialized from integer literal operands. These changes do not need to be repeated for other JIT compilation tiers, and similar changes would be required for any interpreter with a JIT compiler.

JavaScript values As described above, representing `JSValue`s as a union type allowed C++ code to support their size increase with very little modification. Additionally, the fact that NaN-boxed values could be stored in the capability `JSValue`'s address, combined with the fact that 64-bit capability register views operate on capability addresses, meant that JSC bytecode implementations in the LLInt and Baseline JIT with non-pointer operands did not require any modification. These properties are likely to be the similar across contemporary language interpreters.

JSC's particular NaN-boxing representation also composed very well with capability-pointer `JSValue`s. Because pointers are encoded with no modification in JSC's NaN-boxing scheme, `JSValue`s could be dereferenced directly as capability pointers. This means that the pure-capability adaptation did not require have to modify the encoding scheme or add code to unbox or re-derive capability pointers. Other

interpreters that tag or otherwise modify pointers in their representation of JavaScript values may require more significant changes.

Object properties are represented by `JSValue`s in JSC, so the pure-capability adaptation significantly increased the amount of memory required to store JavaScript objects. If the performance effects of this memory increase prove to be too significant, it may be possible to modify the `JSValue` format so that it stores addresses rather than valid capabilities, as will be discussed further below. This kind of change would eliminate the `JSValue` size increase, but it requires a large number of significant source-code modifications and may have a negative effect on security properties. Other interpreters are likely to face a similar trade-off between source-code compatibility and performance in their JavaScript value representations.

Garbage collection Adapting JSC to pure-capability code also facilitated an interesting auxiliary benefit for the garbage collector: its conservative root-scanning algorithm could trivially be made precise. In other words, rather than conservatively assuming every word encountered on the stack that looks like be a pointer actually is one, the garbage collector could check whether a capability pointer under examination is valid, and only add it to the set of root pointers for the marking phase if so. In JavaScriptCore, making the garbage collector precise required adding two lines of code at the beginning of the function `ConservativeRoots::genericAddPointer()`: a check for whether the capability being added is valid, and a return statement to exit the function prematurely if it is not. Without this change, tag violations were observed during garbage collections on Morello. This means that some of the pointers previously being added to the root set were invalid and unnecessary. Other interpreters would achieve similar auxiliary benefits when adapted to pure-capability code: conservative garbage collectors could trivially be made precise, and collectors that are already precise could be made less costly. Independent work that adapted the Boehm-Demers-Weiser Garbage Collector to support pure-capability compilation [53], carried out contemporaneously with the work presented in this dissertation, also identified the benefits of pure-capability compilation for making conservative garbage collectors precise.

4.1.3 Correctness

An important part of assessing the JSC adaptation's feasibility is to verify that the pure-capability version does not change the interpreter's functional behavior relative to the unmodified 64-bit Arm version. To evaluate correctness in this way, I ran the ECMAScript 6 conformance tests contained within the JSC repository. ECMAScript is the name of the JavaScript standard, and ECMAScript 6 is a major release that significantly changed the language [40]. I chose this test suite because JSC implements the entire ECMAScript 6 standard [8], and, while many JavaScript test suites are actually benchmarks that measure performance, this one evaluates standard-conformance across a wide range of features.

I ran the test suite against the unmodified baseline and all three configurations supported in the pure-capability JSC adaptation: LLInt with C++ backend, LLInt with Morello backend, and LLInt with Baseline JIT. Initial discrepancies in the results revealed bugs in the upstream LLInt implementations of the JSC bytecode instructions `is_cell_with_type` and `is_object`. Both implementations incorrectly determined whether a `JSValue` was a heap pointer by checking a mask against its first byte rather than the

entire `JSValue`. These bugs caused integer `JSValue`s with their second bits unset to be interpreted as pointers when passed to the affected operations. While they did not result in faults in the normal 64-bit Arm version of JSC, they manifested as CHERI protection violation faults for trying to dereference an invalid capability on Morello. Fixing each bug required changing the suffix of a single Offline Assembly instruction. With the bugs addressed, the pure-capability JSC configurations achieved parity with the 64-bit Arm version.

I also ran the SunSpider suite of performance benchmarks [110] as a complement to the ECMAScript 6 suite. While they do not explicitly test standards-conformance, they stress-test the interpreter and exercise the JIT compiler. Running this suite on a high-fidelity FPGA model of the Morello SoC uncovered a hardware issue related to tag memory aliasing that was able to be fixed in firmware. The fact that this issue was exposed by JSC workloads indicates that, were it not addressed, it might have been exploitable from untrustworthy JavaScript files controlled by remote attackers. With the issue fixed, all JSC versions ran the benchmarks to completion.

4.2 Source-code compatibility

The amount of source-code change required for an application to support pure-capability compilation affects CHERI's potential for real-world adoption. In this section, I measure the changes that were required for the pure-capability JSC adaptation and compare them to results from prior work on CHERI source-code compatibility. I determine that the source-code compatibility burden of adapting language interpreters to pure-capability CHERI code is not onerous and is comparable to that of similar kinds of software.

In line with previous work on evaluating the compatibility overhead of adapting programs to pure-capability C and C++ [99], I use the `cloc` program [4] to count the lines of code and number of files changed between different JSC configurations. While these metrics have some limitations, they reveal useful information about the developer burden of required source-code changes. The number of lines of code changed is a reasonable proxy for the amount of time it would take a developer to discover and fix issues related to pure-capability compilation, and the number of files changed indicates the scope of the changes within a program's structure.

I configured `cloc` to measure files containing C/C++, Ruby, and assembly code; to consider added, modified, and deleted lines of code; and to ignore blank lines and comments. The baseline configuration was the WebKit 2.27.2 release. I measured changes only in the JavaScriptCore subdirectory of the WebKit project, excluding unrelated browser toolkit components that were not adapted or built pure-capability. The only changes required outside of this subdirectory were in a WebKit utility library that is shared across components. As discussed further below, I handle these changes separately, because it is not straightforward to determine how much of the library is actually being included and built with JSC.

The JSC changeset presented a number of challenges for evaluation that I attempted to address. The changes were research-quality and exploratory, particularly earlier in the project. They included various non-minimized modifications: additions to support debugging, changes to unused code paths, and changes that were required for older versions of the CHERI ISA or CHERI-LLVM compiler but are no longer needed. Additionally, changes to the different execution tiers were intermingled, which made it difficult to

determine how much to attribute to each tier.

To rectify these issues, I created a minimized branch for source-code compatibility evaluation. I removed commits that added features only to support debugging. Each tier of execution had one primary commit that made the majority of the changes to support it, and I edited these to remove debug statements. I also re-ordered commits by the execution tier that they affect. However, a small number of changes attributed to one tier may actually belong to another.

Even with these improvements, there are still some caveats for the minimized changeset. I left some commits with changes that that were required for older versions of the CHERI ISA or compiler but are not required anymore. I also left some commits that touch a significant number of lines but make no functional change; for example, renaming the field of a structure. I included these changes because they might still reasonably be made in practice for clarity or to eliminate compiler warnings.

Additionally, these measurements of the JavaScriptCore source directory include code that supports architectures not targeted or modified in the adaptation, which inflates the codebase size. On the other hand, adding a completely new backend for the LLInt Morello target likely inflates the diff size.

Name	Files modified (%)	Lines modified (%)
to-llint-cloop	45 (1.70)	547 (0.12)
to-llint-morello	112 (4.23)	2641 (0.59)
to-jit-disassembler	153 (5.78)	4176 (0.94)
llint-cloop-commit	14 (0.53)	401 (0.09)
llint-morello-commit	20 (0.76)	1521 (0.34)
jit-commit	31 (1.17)	877 (0.2)
jit-disassembler-commit	3 (0.11)	411 (0.09)
OpenSSH	0 (0.00)	0 (0.00)
rsync	2 (1.90)	3 (0.01)
PostgreSQL	96 (4.94)	803 (0.11)
NGINX	26 (7.78)	145 (0.11)
FreeBSD kernel	590 (7.71)	34105 (1.40)
FreeBSD libc	195 (10.32)	3199 (1.51)
compiler-rt	217 (20.36)	2359 (1.81)

Table 4.1: Source-code compatibility metrics for the pure-capability JavaScriptCore adaptation compared to previous work on other codebases [99]. The compatibility burden of adapting JSC is comparable to that of adapting similar codebases.

Table 4.1 contains compatibility results for the pure-capability adaptation of JSC and compares them to previous work on other codebases. Changes are measured in terms of the absolute number of files and lines added, removed, or modified; they are also presented as a percentage of files and lines in the baseline version of the codebase. The upper segment of the table shows the cumulative changes for supporting different tiers of JavaScript execution. The rows correspond to the JavaScript execution tiers, with the to-jit-disassembler row containing changes required for the Baseline JIT and the disassembler. The middle segment of the table shows changes made by the primary

commits for each execution tier. The primary commits contain almost all of the changes required to support the actual interpreter backends or the JIT, as opposed to changes to auxiliary code paths required for an execution tier to function. In general, the auxiliary changes contributed fewer lines to the overall total but touched more files.

These table segments allow for comparing the relative effort required to support the different execution tiers. The LLInt C++ backend required changing 547 lines, around 400 of which were directly in the LLInt implementation or Offline Assembly backend. Approximately 2100 additional modified lines were required to support the LLInt Morello backend, 1500 of which were direct and 600 of which were auxiliary. A further 1500 changes were required for the JIT compiler and disassembler, with about 900 in the compiler, 400 in the disassembler, and 200 auxiliary.

While changes to support the LLInt Morello backend and baseline JIT were conceptually similar, and the JIT changes included support for assembling and disassembling new instruction encodings, the count of 1500 non-auxiliary lines changed for the LLInt Morello backend was higher than the combined 1300 non-auxiliary lines for the baseline JIT and disassembler. This can be explained by the fact that the LLInt Morello backend used new files that were copied from the 64-bit Arm backend and then modified, while the baseline JIT support modified existing files. It is likely that the LLInt Morello backend could have been supported with fewer changes than the Baseline JIT if it did not make copies of existing files.

Another way to measure the effort of adapting different JSC components is to consider the relative weight of changes in different programming languages. Of the 4176 cumulative lines changed to support the JIT compiler and disassembler, 2559 or 61.28% were in C/C++, 1273 or 30.48% were in Ruby, and 344 or 8.24% were in assembly. The C/C++ changes were for the JIT and all auxiliary code, the Ruby changes added the Morello Offline Assembly compiler backend and modified the C++ one, and the assembly changes were modifications to the Offline Assembly LLInt implementation.

The lower segment of Table 4.1 contains data about the number and percentage of file and code modifications required for pure-capability support in other codebases. There are three kinds of codebase represented. The first consists of applications like OpenSSH and rsync that use pointers in a standards-compliant way and need no or very little modification for pure-capability support. The second comprises high-performance applications like PostgreSQL and NGINX that require slightly more modification: about 5% of files and 0.1% of lines of code. The third are operating-system-level components, such as FreeBSD's kernel and libc and LLVM's compiler runtime library. These require modification to over 5% of files and about 1.5% of lines of code.

The results from other codebases reveal that JSC's compatibility burden does not exceed that of other types of software. Support for the LLInt C++ backend has a compatibility profile similar to that of applications like PostgreSQL and NGINX. Percentage-wise, cumulative changes required for the Baseline JIT and disassembler are significantly higher, but still lower than the changes required for OS-level components. These results are reasonable, considering that the LLInt C++ backend operates like a sophisticated C application, and the other execution tiers add compilation features that are more characteristic of OS-level components.

As mentioned above, some changes were required in a WebKit utility library that I did not consider in the calculations of Table 4.1. I chose not to include the library because, while it contained about 105000 lines of code in total, it was not straightforward

to measure the number of those lines that were actually being included and built as a part of JSC, and I wanted to avoid artificially diluting overhead percentages. Approximately 100 lines had to be changed in the utility library cumulatively to support the Baseline JIT and disassembler. This represents only 0.02% of the 444499 lines in JSC. Significantly overestimating the utility library's impact by adding 0.02% to the percentage of lines modified for each cumulative measure does not meaningfully change the results.

As a whole, these data can be considered as measurements from a not-entirely-minimized changeset to support pure-capability compilation for a contemporary language runtime with JIT compiler. Their costs are reasonable relative to other codebases, with the LLInt C loop backend roughly similar to a high-performance database or web server and the Baseline JIT and disassembler between a sophisticated application and OS-level code.

4.3 Security

In this section, I consider the security benefits of pure-capability JSC and how they compare to those of a typical pure-capability application. I identify some shortcomings and then demonstrate how additional changes can bring pure-capability JSC up to par. Finally, I investigate how the pure-capability adaptation interacts with some of JSC's existing security mitigations.

4.3.1 Pure-capability JSC

In Chapter 3, I defined an analytical framework for reasoning about the security properties of memory-safety mitigations, and I applied it to analyze the CheriABI protections that typical applications receive when they are adapted to support pure-capability compilation. Here I apply the same framework to pure-capability JavaScriptCore in order to highlight any differences.

TCB Pure-capability JSC has a similar, but not identical, TCB to a typical pure-capability application. Because JSC uses a custom heap allocator, it does not rely on the standard library allocator to enforce heap spatial safety. As a result, JSC's heap allocator is part of its TCB, and the standard library one is not.

Additionally, the LLInt and JIT compilers must be added to the TCB in order to support spatial safety on the stack during JavaScript execution. Recall that the JavaScript execution tiers are not implemented in C/C++ and use a custom ABI. As a result, the LLInt and JIT compiler implementations have direct access to the privileged stack pointer capability, and they must be trusted not to use that capability to violate the bounds of memory objects allocated on the stack.

The other components of a typical pure-capability application's TCB, namely the CHERI-LLVM compiler, the CheriBSD kernel, and the runtime linker, are also part of the TCB for pure-capability JSC.

Threat model The threat model for pure-capability JSC is identical to the basic threat model for typical CheriABI applications. In the specific case of JSC, the attacker-controlled unprivileged and untrustworthy input stream is the JavaScript source being executed, and JSC is the non-malicious but potentially buggy program executing it. However, pure-capability JSC does not support the expanded CheriABI threat model that takes

inter-DSO compartmentalization into account. As will be discussed further below, without additional changes pure-capability JSC does not support implementing return addresses as sentry capabilities, which limits the effectiveness of inter-DSO compartmentalization. Furthermore, JSC is a monolithic DSO that contains the implementation of its own heap allocator, which must be considered a part of the TCB in order to support heap spatial safety.

Memory protections Unlike typical pure-capability applications with CheriABI protections, pure-capability JSC does not completely eliminate spatial memory-safety vulnerabilities. By default, its custom heap allocator does not set fine-grained bounds on allocations. Capabilities returned by the allocator have bounds, set as a result of the allocator's call to `mmap()`, that span their containing 16KB slab. Furthermore, as described above, the LLInt and JIT compilers do not guarantee stack spatial safety for JavaScript execution by default. However, pure-capability JSC does mitigate the same attack mechanisms as a typical CheriABI application: code injection, code pointer injection, and data pointer injection.

4.3.2 Parity with CheriABI

The above analysis demonstrates that pure-capability JavaScriptCore does not automatically have the same security properties as a typical pure-capability CheriABI application. In this subsection, I describe the changes required to bring them up to par.

Heap spatial safety Adding support for heap spatial safety in JSC is surprisingly straightforward due to the design of its heap allocator. All normal allocations pass through the `FreeList::allocate()` function, which contains three code paths that can return a cell. Enforcing fine-grained bounds for memory objects on the heap simply requires adding a single line that sets the bounds of the capability being returned according to the containing slab's cell size for each code path.

Because the heap is slab-allocated using free lists, capabilities to cells do not need to be re-derived with expanded bounds when they are freed by the garbage collector. The only part of JSC that requires the re-derivation of heap capabilities are the functions `MarkedBlock::blockFor()` and `HandleBlock::blockFor()`. These functions take a pointer to a heap cell and return a pointer to the beginning of its 16KB block, or slab. On a non-CHERI architecture, it can mask the address of the passed-in cell pointer, but this is not possible with capabilities. Instead, a capability to the block must be re-derived from a more privileged capability. This re-derivation requires traversing bookkeeping structures internal to the heap allocator so that privileged block capabilities returned from `mmap()` can be retrieved as necessary.

LLInt and JIT compilers In order to guarantee spatial safety on the stack for JavaScript execution, the LLInt and JIT compiler implementations must be shown not to use the privileged capability stack pointer to violate the integrity of memory object bounds. Uses of the stack pointer in these environments can be audited and either statically determined not to violate spatial safety or augmented to derive and use a separate capability with fine-grained bounds. These are analogous to changes made in the CHERI-LLVM compiler to support spatial safety on the stack for C and C++. As a form of defense in depth,

it is also possible to bound the capability stack pointer on transitions between the C++ codebase and JavaScript execution in order to guarantee that JavaScript execution cannot interfere with local variables defined in C++.

Sentry return addresses Return addresses cannot be implemented as sentry capabilities by default in pure-capability JSC. JSC's JIT compilers rely on the ability to dereference return addresses in two places. One validates code pointers at call boundaries between JIT-compiled code and the C++ runtime by performing a one byte read from them. The other involves using the return address of a JIT-compiled function to re-link its call site to a different function. Both of these actions cause pure-capability JSC to crash when return addresses are implemented as sentry capabilities.

Modifying JSC to support implementing return addresses as sentry capabilities involved straightforward changes to these two code paths. The code that validates a return address by reading a byte from it can be removed when targeting CHERI architectures, as the check is not actually required for correctness. To support the re-linking of call sites for JIT functions via their return addresses, the JIT compiler can re-derive an unsealed capability referencing the return address from a privileged capability to the entire JIT executable memory region that it maintains internally. I describe the JIT executable memory region in more detail below.

Inter-DSO compartmentalization JSC is currently built as a single monolithic DSO that contains components of its TCB, including its heap allocator, interpreter, and JIT compilers. In order to support the expanded CheriABI threat model, JSC can be split into multiple DSOs and benefit from the inter-DSO compartmentalization provided by pure-capability linkage. Future work on explicit fine-grained compartmentalization with CHERI may allow JSC to be suitably compartmentalized while remaining a single DSO.

4.3.3 Existing mitigations

In this subsection, I examine how the pure-capability adaptation of JSC interacts with some of its existing security mitigations. Some mitigations are rendered unnecessary with CheriABI-level protections, while others are incompatible with the semantics of capability pointers.

JIT executable memory allocator A JIT compiler, by definition, must be able to write code to memory and then later cause it to be executed. However, having a region of memory that is constantly mapped with both write and execute permissions opens the door for attackers to do the same. In JSC, executable memory used by the JIT compiler is managed by a special allocator, and steps are taken to prevent attackers from abusing executable memory.

More specifically, memory managed by the executable allocator is mapped twice: one mapping has read and execute permissions, and the other, mapped at a random address, has write permissions. A single pointer to the writable mapping is saved in an unreadable, execute-only stub function that can be called to copy JIT-ed code into the executable memory. To access this stub meaningfully, attackers would have to hijack control flow to call it with controlled arguments, which is much more difficult than writing to an executable region of memory directly. However, this mitigation is probabilistic, because

an attacker could theoretically guess the address of the mapping with write permissions and inject code directly there. When targeting recent versions of iOS, JSC makes the mitigation deterministic by augmenting the scheme with a system register that protects the writable mappings. The memory-copying stub function toggles the register to enable and disable write access before and after writing to the memory, and no other code paths enable writes.

Pure-capability compilation provides an alternative, straightforward route to a deterministic mitigation. All capabilities returned by the executable memory allocator can have their write permissions cleared, with only the memory-copying stub function retaining a privileged write-enabled capability to the region. This design has the same effect as the deterministic mitigation described above, and it does not require hardware support beyond CHERI.

Structure ID randomization Recall from Section 2.6 that JSC uses the `Structure` class to map between a JavaScript object's property names and storage locations. Pointers to these `Structure`s are dynamically added to a global table as new JavaScript object shapes are created. Each `JSCell` has a field holding its Structure ID, which is an index into the table.

A potential weakness with the above design is that attackers can construct fake JavaScript objects on the heap by spraying and guessing Structure IDs that correspond to a known JavaScript object shape. In an attempt to mitigate this possibility, JSC adopted Structure ID randomization, which performs an exclusive-or of the `Structure` pointer in the table with some randomly generated bits that are accessible to legitimate code paths but unknown to the attacker. Without knowledge of the random bits, the attacker should be unable to reconstruct a usable `Structure` pointer.

Because Structure ID randomization involves scrambling pointers, it is not compatible with capability-pointer semantics and must be disabled in pure-capability JSC. However, the mitigation is probabilistic and can be bypassed easily [49]. The security benefits of pure-capability JSC outweigh the costs of disabling Structure ID randomization.

Heap isolation JSC has adopted a number of software-based mitigations for isolating memory objects on the heap. For example, Gigacages are separate heaps, distinct from the primary heap and from each other, that allocate particular types of memory object [45]. They are designed to limit memory-safety issues related to objects within a Gigacage from affecting objects outside of it. They make exploitation more challenging, but they have been bypassed and come with a significant performance cost: the Gigacage for JavaScript strings was removed because the security benefit was determined not to be worth the performance degradation [93]. Another similar feature is the IsoSubspace, a segregated section of the primary heap that is only used to allocate objects of similar size.

While CheriABI-level protection for JSC guarantees spatial safety vulnerabilities on the heap, the above mitigations attempt to handle spatial-, temporal-, and type-safety issues. If CHERI-based mitigations for temporal and type safety were applied to JavaScriptCore, they might obviate the need for software-based heap isolation and yield performance improvements. Investigating this possibility is left as future work.

Pointer authentication On relatively new devices running iOS, JSC uses Arm's Pointer Authentication Codes (PAC) [103] to provide a form of control-flow integrity. The PAC

mitigations are technically probabilistic, and they have been bypassed in multiple published attacks [10, 49]. Pure-capability JSC's prevention of pointer injection offers similar, but deterministic, control-flow integrity properties.

4.4 Discussion

The above analysis suggests that it is indeed feasible to adapt a contemporary language interpreter with JIT compilation to pure-capability code. While JavaScriptCore is a large and complex codebase, and significant changes were required to add pure-capability support, most of those changes were related either to JavaScript execution, `JSValue` representation, and C++ pointer use. And, with an understanding of the incidental complexity of JSC's codebase, the changes within these categories were similar and largely straightforward. The resulting adaptation did not exhibit functional changes relative to the normal 64-bit Arm version. These characteristics are likely to apply to other similar interpreters.

In terms of source-code compatibility, the burden of adapting JSC to pure-capability code is comparable to that of similar kinds of software. Support for the LLInt C++ backend has a compatibility profile similar to that of sophisticated C++ applications like NGINX and PostgreSQL, and supporting the Baseline JIT and disassembler is similar to adapting OS-level components.

Because of its unusual features, pure-capability JSC does not automatically have the same CheriABI-level security properties that typical applications obtain when they are adapted to support pure-capability compilation. However, a few straightforward changes can bring its security properties up to par. And while pure-capability compilation requires that certain ineffective existing mitigations in JSC be disabled, it offers opportunities to subsume the functionality and reduce the performance cost of others.

4.4.1 Future work

Though significant, the work presented in this chapter was early-stage and subject to time and platform limitations. It leads naturally to a number of directions for future work. These include:

Performance measurement An important piece of future work, once current limitations of the Morello SoC platform have been addressed, is to measure the performance cost of pure-capability JSC. Doubling the pointer size will certainly increase JSC's cache footprint and memory accesses, which can have significant effects on throughput and energy consumption. The extent of these effects will determine whether pure-capability JSC can reasonably be adopted in practice.

Integer pointers to the JavaScript heap One possibility for reducing the memory overhead of capabilities in JSC would be to represent pointers in the C++ codebase as capabilities, but to keep `JSValue` pointers to the JavaScript heap as 64-bit integers. More specifically, the JavaScript heap could be allocated from a contiguous region of memory, which could be spanned by CHERI's default data capability (DDC). Integers could then be used as offsets against the DDC to access memory within the region.

Such a scheme would reduce memory overhead relative to pure-capability JSC, and using DDC to perform integer-pointer offsetting from a base requires fewer instructions than doing so on a traditional architecture. However, using integer pointers for the JavaScript heap also comes with challenges. Most directly, it would mean that the heap no longer benefits from capability protections. Additional analysis will be required to determine the full effects of this security degradation.

Another challenge related to using integer pointers for the JavaScript heap is an increased source-code compatibility burden. An exploratory changeset to support integer pointers in execution up to the Morello LLInt backend has been developed on top of the adaptation described in this chapter. Among other changes, it involved making the JavaScript heap contiguous; decorating all pointers to the heap with a C++ class that allows their implementation to be switched between capabilities and integers at compile time; updating Offline Assembly and the LLInt implementation to understand `JSValue`-sized operands as distinct from pointer and 64-bit operands; and tagging `JSValue` operands so that alternate-base Morello instructions are emitted as appropriate.

In general, these changes touched a significant amount of code: about 2400 lines and 135 files. On the other hand, they render some of the modifications made to support capability pointers to the heap unnecessary. For example, structure and heap pointer packing techniques do not need to be disabled, and bitwise operations on `JSValue` pointers do not need to be modified or annotated.

Additional JIT execution tiers The pure-capability JSC adaptation should also be extended to support the optimizing DFG and FTL JIT execution tiers. This would allow for further feasibility, source-code compatibility, and performance analysis. It would also facilitate the investigation of JSC features that have not yet been adapted to support pure-capability compilation, such as on-stack replacement.

Temporal safety Although JavaScript is a garbage-collected language, logic bugs in the garbage collectors of JavaScript interpreters still allow for the possibility of heap temporal-safety vulnerabilities, as demonstrated by the JSC 2 and JSC 6 exploits analyzed in Section 3.4.1. To remedy this problem, novel sweeping capability revocation techniques to eliminate heap temporal safety vulnerabilities, described in Chapter 5, could be investigated in the context of JSC.

Chapter 5

Heap temporal safety

One way to guarantee heap temporal safety in manually managed languages such as C and C++ is to ensure that, before a region of heap memory is reallocated, all outstanding pointers to it have been invalidated, or rendered unusable. With traditional architectures, performing this invalidation requires expensive software instrumentation that comes with unacceptable overheads [76, 111]. However, capability architectures like CHERI, which identify and bound pointers architecturally, present promising opportunities for a low-cost form of pointer invalidation.

In a capability architecture, invalidating a pointer to the heap is equivalent to revoking a capability that grants access to a particular region of heap memory. Past capability systems have implemented capability revocation by indirecting capability accesses through a central lookup table entry that can be invalidated in a single operation [98]. However, this indirection adds overhead to all memory accesses, which is incompatible with CHERI's design goals. A new technique is thus needed for capability revocation with CHERI.

A previous paper described CHERIvoke [127], an algorithm for *sweeping CHERI capability revocation*. The main principle behind sweeping capability revocation is that, before a region of the heap can be reused, all of memory must be scanned for capabilities granting access to the region, and all such capabilities found must be invalidated. The CHERIvoke paper presented an outline of the sweeping revocation algorithm, and ran simulations on contemporary x86 hardware to estimate its performance, but did not implement or evaluate it on CHERI systems.

In this chapter, I describe the design, implementation, and evaluation of a fully elaborated system for CHERI capability revocation that guarantees heap temporal safety for userspace C/C++ programs. The system consists of Cornucopia, a CheriBSD kernel subsystem that provides a capability revocation service to userspace programs, as well as userspace allocators that have been adapted to use the Cornucopia interface. Cornucopia can be configured to use the CHERIvoke algorithm for sweeping revocation or to use one of two novel algorithms that introduce support for concurrency and have improved performance characteristics. These algorithms, inspired by prior work on garbage collection, make use of new architectural features implemented in Morello to support capability store-side and load-side barriers. I also introduce and evaluate a generic wrapper module for userspace heap allocators that allows them to take advantage of Cornucopia with minimal source-code changes required relative to direct integration.

I aim to address the following questions in this chapter:

- Is it possible to implement sweeping capability revocation on real CHERI hardware

to mitigate heap temporal safety vulnerabilities in practice?

- Do concurrent sweeping revocation algorithms using store-side and load-side barriers significantly improve performance relative to the CHERIvoke baseline?
- Does using the generic wrapper module to integrate userspace heap allocators with Cornucopia offer significant source-code compatibility benefits at a reasonable performance cost relative to direct integration?

I determine that they can all be answered affirmatively.

5.1 Introduction

This section introduces and provides context for the novel work presented later in the chapter. It first describes how guaranteeing heap temporal safety can be formulated as a problem of capability revocation, as well as how revocation has been implemented in past capability systems. It then highlights the features of CHERI that make it amenable to sweeping revocation. It summarizes the CHERIvoke [127] algorithm for sweeping CHERI capability revocation and presents a high-level overview of how heap temporal safety can be guaranteed using the Cornucopia subsystem of the CheriBSD kernel.

5.1.1 Heap temporal safety as capability revocation

In Section 2.1, I defined a C/C++ execution environment to be temporally safe if it guarantees the integrity of memory object lifetimes. I now consider in more detail what this definition means for userspace heaps.

Temporal-safety concerns also apply to the stack, whose memory objects have dynamic lifetimes. However, in line with prior work on temporal safety [2, 30, 64], I consider the stack out of scope. References to stack memory objects are amenable to static escape analysis [30], and objects with possibly escaping pointers could be allocated on the heap.

References to heap objects proliferate throughout a system: they can end up in registers, global data, on the stack, on the heap, and even in the kernel. As a result, after a programmer calls `free()` to release some region of heap memory, pointers to the region might remain accessible. Accessing a freed object is undefined behavior in C and C++ [95, 126], and it indicates a heap temporal safety vulnerability in the program. However, these types of errors happen frequently in practice, especially in large and complex codebases.

A key observation about heap temporal-safety vulnerabilities is that they are only exploitable if the memory underlying the accessible freed object is subsequently used to store some other memory object. In this case, the two memory objects are said to *alias*, i.e. they are backed by the same memory.

For example, suppose a memory object of type a is freed at location L on the heap, but a *dangling pointer* to the freed object remains usable. An object of type b is then allocated at location L. The memory objects of type a and b alias, since there are usable references to both objects that point to L. An attacker that controls program inputs, e.g. a script being executed by a JavaScript interpreter, might then use a code path that references L as an object of type a to corrupt a field in the structure defined by type b. The corrupted field might influence the behavior of a code path that references L as an object of type

b. On non-CHERI systems, the attacker may be able to corrupt a code pointer that is subsequently the target of a jump, granting them control over code execution.

On the other hand, if a heap temporal-safety bug results in an accessible dangling pointer to some freed memory object, but the underlying memory does not alias with any other memory objects, then the bug is not exploitable. The memory might be accessed after it is freed, but only by code paths that operate on the original memory object.

Heap temporal-safety vulnerabilities in C and C++ are often called use-after-free or double-free vulnerabilities, depending on how they manifest in the program. As the above discussion demonstrates, however, exploitable heap temporal-safety vulnerabilities could more accurately be described as use-after-reallocation vulnerabilities, because they involve aliasing when memory underlying a heap allocation is reallocated for some other purpose. Note that this aliasing might be between two heap-allocated memory objects or, if the allocator uses freed memory for bookkeeping, between a heap memory object and allocator-internal metadata.

To mitigate heap temporal-safety vulnerabilities completely, heap memory object aliasing must be made impossible. As described in Section 2.1.2.2, one technique for achieving this goal is pointer nullification, also known as pointer invalidation. A pointer nullification scheme prevents aliasing by guaranteeing that, before any heap allocation is considered free for reuse by a memory allocator, all outstanding pointers to that allocation have been rendered unusable.

An important additional feature that must be composed with pointer nullification schemes to guarantee heap temporal safety is the zeroing of heap memory before it is reused. Otherwise, uninitialized reads from reallocated memory could result in a form of aliasing.

Past approaches to pointer nullification on commodity architectures have relied on software instrumentation to track the location and movement of pointers in memory, which can be inexact and result in significant overheads [61, 64, 76]. A performance overhead of under 5% has previously been identified as necessary to the facilitate universal adoption of a memory safety mitigation [111], and software approaches have not been able to achieve this goal.

Using capability hardware to support pointer nullification is a promising way to facilitate performance and security improvements. In capability systems, nullifying a pointer is equivalent to revoking a capability, i.e. rendering the capability invalid and unusable. A capability revocation scheme for heap temporal safety guarantees that, before a region of memory is considered free for reuse by a memory allocator, no valid capabilities to the region exist outside of the allocator.

5.1.2 Sweeping capability revocation

A fundamental property of capability systems is that capabilities can be copied and propagate freely throughout memory. This property makes the provision and use of capabilities straightforward, but it poses challenges for revocation. Historic capability systems commonly used indirection to solve the revocation problem [98]. More specifically, using a capability to access memory required looking up its corresponding entry in a protected table. All copied and derived capabilities shared the same table entry, so an application program's access to a region of heap memory could be revoked in a single operation by clearing the appropriate table entry.

However, while indirection-based techniques make capability revocation simple, they have significant drawbacks. Most prominently, they add overhead to memory accesses, which are more common than revocation events. Today's systems already perform table lookups on memory accesses to support virtual memory; adding another synchronous table lookup goes against the principles of contemporary microarchitecture.

CHERI avoids additional indirection and table lookups by design [118], and each CHERI capability has its own authorization to access memory directly. Thus, indirection-based revocation is not appropriate for CHERI systems. An alternative technique known as sweeping capability revocation is more suitable. In a sweeping capability revocation system, access to a given region of memory is revoked by a routine that scans an application's entire memory and revokes all encountered valid capabilities that reference the given region.

A naïve implementation of sweeping capability revocation to guarantee heap temporal safety could perform a sweep of memory on each call to `free()` that revokes all outstanding capabilities to the passed-in region before the allocator considers it available for reuse. While this approach would incur prohibitive performance costs, it is amenable to significant optimization in practice. CHERI's architectural features make sweeping capability revocation not only possible, but also less reliant on expensive software instrumentation than past solutions for heap temporal safety on commodity architectures, which were discussed in Section 2.1.2. The benefits of CHERI's design in supporting sweeping capability revocation include the following:

Spatial safety with CheriABI Sweeping capability revocation can build on top of the spatially safe CheriABI process environment. All pointers are implemented as capabilities that restrict access to their referenced memory objects. In particular, the heap allocator applies fine-grained bounds to capabilities that it grants to applications. CHERI capability bounds are monotonically decreasing, which means that the bounds of a derived capability must be a subset of the bounds of the capability it was derived from. As a result, any valid and usable capability that has been derived from a capability to some heap allocation unambiguously identifies that allocation by its bounds, even if its bounds have been shrunk or its address points out of bounds. This property is sufficient for the revoker to identify all capabilities that must be revoked when a particular region of memory is freed.

Revoker-visible tags The fact that all usable pointers are represented as tagged capabilities in a CheriABI process means that pointers can be identified precisely by the revoker at no additional performance cost. On traditional architectures, tracking pointers precisely requires software instrumentation, and heuristic pointer identification can generate unnecessary work. CHERI capability pointers also cannot be obfuscated to escape detection or forged. Even hand-written assembly code must follow capability rules. These properties are crucial for correctness and differentiate sweeping capability revocation from past techniques for heap temporal safety in C/C++ on commodity architectures.

Architecture-visible tags Because capability tags are architecturally visible, the architecture can specify functionality that discriminates between pointers and data. For example, permission bits could be added to MMU page table entries that cause faults on pointer loads or stores, but not data loads or stores. Such features are

not possible in traditional, untagged architectures. They facilitate optimizations to sweeping capability revocation, as I discuss further below.

As a technique for heap temporal safety, sweeping revocation is superficially similar to, but ultimately different from, mark-and-sweep and copying garbage collection, which I discussed in Section 2.8. Such garbage collectors obviate the need for manual calls to free heap memory. They first traverse reachable memory objects to identify heap regions to which there are no outstanding references, and then scan memory to reclaim those regions for reuse. On the other hand, sweeping revocation uses manual `free()` calls to identify regions that should be made available for reuse, and scans memory to revoke any outstanding references to those regions. While they have significant differences, both garbage collection and sweeping capability revocation involve a sweep of memory. Prior work on optimizing garbage collectors [13] inspired optimizations for sweeping capability revocation, including the novel designs described in this chapter.

5.1.3 CHERIvoke

CHERIvoke [127] is an algorithm for heap temporal safety with sweeping CHERI capability revocation. Relative to the naïve approach of sweeping all of memory on every call to `free()`, it makes some optimizations and suggests architectural support to improve performance. It was evaluated in simulation by implementing an approximation of the algorithm on contemporary x86 machines, and its performance overheads were found to be significantly lower than other state-of-the-art systems for heap temporal safety. However, it was not implemented on CHERI systems.

CHERIvoke serves as a baseline for the novel work presented later in this chapter. Its primary features are as follows:

Quarantine buffer Rather than performing a revocation pass, i.e. a sweep of memory, for each freed heap region, CHERIvoke aggregates freed regions and performs one pass for an entire batch. With CHERIvoke, when a heap allocation is passed to `free()`, the allocator places it in a quarantine state to indicate that it is not safe for reuse and adds it to a list of such regions that make up the *quarantine buffer*. When the quarantine buffer becomes full, a single sweep of memory revokes all outstanding valid capabilities to any region contained in it, and then the regions in the buffer are returned to the free state and made available for re-allocation. The size of the quarantine buffer can be configured as a static number or as a percentage of the current heap size. The higher the quarantine buffer size, the more the cycle cost of sweeping memory gets amortized. However, the application's memory footprint also increases with the quarantine buffer size.

Shadow bitmap With multiple freed regions of heap memory being aggregated in the quarantine buffer, there must be a scalable way to tell the revoking routine precisely which regions of memory are subject to revocation. CHERIvoke uses a shadow bitmap, which is a region of application memory that contains one bit corresponding to each 16-byte region of the application's entire address space, for this purpose. For example, a 16-byte heap allocation corresponds to one bit in the shadow map, a 32-byte heap allocation corresponds to two bits, and so on. 16 bytes is the minimum heap allocation granularity and alignment on CHERI systems, so this granularity is sufficient to represent all heap allocations.

Before a revocation pass, the allocator sets bits in the shadow bitmap corresponding to all of the memory regions that have been added to the quarantine buffer. Then, during the pass, each time the revocation routine encounters a valid capability, it checks the bit corresponding to the capability's base in the shadow bitmap. If that bit is set, it revokes the capability. It checks the bit corresponding to the capability's base because, as discussed above, capability bounds are monotonically decreasing, so the base of a valid capability to the heap must be within the bounds of its initial heap allocation, even if the capability has otherwise been modified. After the revocation pass is complete, the revoker clears bits corresponding to the processed quarantine buffer in the shadow bitmap.

Capability-dirty tracking An architecture-supported optimization in CHERIvoke is capability-dirty tracking, which tracks the flow of capabilities through memory and allows the revoker to avoid scanning pages that are known not to contain any capabilities. The term capability-dirty indicates a mapping that has experienced a capability store. This is analogous to the commonly used term dirty, which indicates a mapping that has experienced any kind of store. It involves the use of a new abstract PTE bit, `CapDirty`, which, when not set for a particular PTE, causes stores of capabilities via the mapping to result in a fault. Empty pages are initially mapped with `CapDirty` cleared. On the first capability store to the page, the system uses the resulting fault to set `CapDirty`, and the capability store can proceed. Support for the `CapDirty` bit allows the revocation routine to avoid scanning pages of virtual memory that have mappings with the `CapDirty` bit clear. The `CapDirty` bit can result in false positives, because removing the last valid capability from a page does not automatically clear the bit in corresponding PTEs. However, when the revocation routine scans a virtual page that is found not to contain any capabilities, it can clear the corresponding `CapDirty` bit. For most of the benchmarks evaluated in the CHERIvoke paper, capability-dirty tracking reduced the amount of memory that needed to be swept by over 40% [127].

CLoadTags Another architectural optimization in CHERIvoke allows the revocation routine to examine capability tags without loading their associated data. Previously, to determine whether a word of memory was a tagged capability or not, the CHERI architecture required loading the word into a register and using an instruction to check the value of its tag bit. This approach not only has a high computational cost, but it also increases DRAM traffic and pollutes the cache hierarchy. CHERIvoke introduces the `CLoadTags` instruction, which loads the tags for a region of memory without loading the associated data. For example, if a 64-byte region starting at some address A contained two tagged capabilities, one at offset 0 and one at offset 16, the `CLoadTags` instruction with operand A would return 0011. A revocation routine making use of this instruction would then only have to load the two words known to contain capabilities and look up their bases in the shadow bitmap.

This optimization reduces the computational overhead of revocation by allowing the revoker to skip regions and words known not to contain capabilities. It also reduces DRAM traffic and cache pollution. In the implementation suggested by the CHERIvoke paper, `CLoadTags` returns tags from a data cache if the requested region is already present in some cache line; otherwise, it returns tags directly from the tag controller and does not cache the result. Thus, the contents of a region of memory

that contains no capabilities does not need to be loaded into a data cache. The amount of skipped memory can be significant: analysis in the CHERIvoke paper found fewer than one quarter of cache lines hold pointers for many applications [127].

The CHERIvoke sweeping routine, taking the above features into account, is described by the pseudocode in Figure 5.1. Note that the performance cost of a revocation pass is roughly independent of the amount of address space that is set in the shadow bitmap, i.e. the amount of memory for which corresponding references should be revoked. It is also roughly independent of the number of found capabilities that are actually revoked. The cost of a pass is primarily determined by the size of the application's memory and the distribution of all valid capabilities, whether subject to revocation or not, within it. The rate at which heap memory is freed determines how frequently revocation passes will take place. Crucially, the CHERIvoke algorithm does not support concurrency. It assumes that the application is paused for the entire revocation pass.

```
1  foreach page in application_pages
2    if !has_capdirty_mapping(page)
3      // no capabilities present on page
4      continue
5
6    // i iterates over CLoadTags regions in a page
7    // j iterates over capability-sized slots in a region
8    for (i = page; i < page + PAGE_SIZE; i += CLT_REGION_SIZE)
9      tags = CLoadTags(i)
10     for (j = i; tags != 0; tags >>= 1, j += CAP_SIZE)
11       if (tags & 1)
12         // CLoadTags indicated a valid capability here
13         valid_capability = load_capability(j)
14         if shadow_bit_set(get_base(valid_capability))
15           // the capability references a region
16           // marked for revocation
17           revoke(valid_capability)
```

Figure 5.1: Pseudocode for the CHERIvoke algorithm's revocation pass.

5.1.4 Temporal safety with Cornucopia

Cornucopia is a novel, fully elaborated kernel subsystem for sweeping revocation that has been implemented in CheriBSD. Relative to CHERIvoke, it introduces new sweeping algorithms that support concurrency and feature improved performance characteristics. It also addresses design challenges that must be dealt with in a real system implementation. Mitigating userspace heap temporal-safety vulnerabilities with Cornucopia involves cooperation between the kernel subsystem and userspace memory allocators. Allocators communicate with Cornucopia via a shadow bitmap and system calls.

Cornucopia's ultimate goal is to facilitate low-cost, always-on mitigation for heap temporal-safety vulnerabilities. It is a fail-closed system that faults on attempted use of a revoked capability. It does not aim to be a sanitizer or debugging aid [106], and so does not

detect harmless uses of memory that has been freed but not yet been subject to revocation. It also cannot detect the use of integer data derived from capabilities. For example, a map keyed by the addresses of heap memory objects might return data corresponding to a previously freed object even after a revocation pass.

In the following section, I describe the Cornucopia kernel subsystem and its algorithms for concurrent sweeping revocation. In Section 5.3, I demonstrate how to adapt memory allocators to make use of Cornucopia features, which was not addressed in the CHERIvoke paper. In Section 5.4, I evaluate the mitigation of heap temporal-safety vulnerabilities with Cornucopia according to the hypotheses laid out at the beginning of this chapter.

5.2 The Cornucopia kernel subsystem

Cornucopia implements sweeping revocation as a kernel service that can be invoked by userspace processes. Implementing sweeping revocation in the kernel is a natural choice because of the privilege required to perform revocation passes. For example, the shadow bitmap interface between a userspace heap allocator and the revoker should be mapped at a consistent and contiguous region of each process' virtual address space. This is most straightforwardly achieved by the kernel. Additionally, the revoker needs to examine all of a process' memory; adding a userspace component with the ability to do so goes against the principle of least privilege and CheriABI's bounds minimization.

Implementing revocation in the kernel is also necessary for correctness. Capabilities from a userspace application are often passed to the kernel as system-call parameters, and these must be processed by the revoker in order to completely rule out temporal-safety violations. Capability arguments for typical system calls, such as `read()`, are only present in the kernel for the time it takes to execute the system call. The revoker handles these by ensuring that, when an application gets paused during revocation, its threads are at a system call boundary; in other words, no system calls are in progress, and userspace capabilities passed to typical system calls reside in trap frames that the revoker can examine.

However, some system calls hold userspace capabilities in data structures after their invocation. An example of these are `kqueue` triggers, which save a userspace pointer in the kernel and return it to the application in response to some event. System calls for asynchronous I/O similarly retain userspace capabilities. For these cases, the revoker does not scan all of kernel memory to detect capabilities subject to revocation. Capabilities from many processes are stored in the kernel, and it is not straightforward to identify the userspace process corresponding to a capability observed in memory in general. Instead, the revoker includes specific routines to handle each subsystem that might persist userspace capabilities. For each subsystem, it identifies data structures for the process being revoked and then checks those data structures for relevant capabilities.

In this section, I first describe the overall design of the Cornucopia kernel subsystem and its interface with userspace processes. I then focus on the design and implementation of the novel store-side and load-side barrier algorithms for concurrent sweeping capability revocation. Cornucopia can be configured to use one of these new algorithms or a concrete implementation of the CHERIvoke algorithm.

5.2.1 Subsystem design

The Cornucopia subsystem interfaces with userspace via shared memory for a shadow bitmap and system calls. The primary features of the interface are as follows:

Shadow bitmap Cornucopia uses the shadow bitmap design described in the CHERIvoke paper, with one bit per 16 bytes of the entire virtual address space, to track regions of memory for which corresponding capabilities should be revoked. It reserves the same contiguous region of each process' virtual address space to store the map. Allocators obtain capabilities to the shadow bitmap via the `cheri_revoke_shadow()` system call, which accepts a capability to a region of userspace memory and returns a capability to its corresponding region of the shadow bitmap.

This system call requires that the input capability has the `VMMAP` permission set. `VMMAP` is set only for capabilities returned from the `mmap()` system call; a `VMMAP`-bearing capability indicates that the software component holding it has been granted ownership of the corresponding region of virtual memory by the kernel or some delegated authority. CheriABI heap allocators maintain internal copies of privileged `VMMAP`-bearing capabilities returned by `mmap()`, but they clear the permission on any capability that is returned to the application. This guarantees that unprivileged userspace components cannot gain access to regions of the shadow bitmap that correspond to the heap. In other words, they cannot interfere with the revocation of capabilities that point to the heap. This design also supports multiple potentially distrusting allocators within a single process.

Revocation API Cornucopia adds the `cheri_revoke()` system call, which allows userspace processes to trigger a revocation pass. This system call uses the calling thread to perform revocation. In other words, it performs an entire revocation pass before returning execution back to userspace. If the revocation algorithm being used supports it, other application threads can proceed concurrently with the revocation pass, except during stop-the-world phases, as discussed further below. Single-threaded userspace processes can benefit from concurrent revocation by using a dedicated worker thread to make the `cheri_revoke()` system call.

`cheri_revoke()` allows the caller to select which sweeping revocation algorithm to use. It also fills out a caller-supplied structure with statistics about the revocation pass, and it supplies information about the revoker's internal state.

Revocation The Cornucopia revoker performs revocation sweeps by examining memory directly from the supervisor context. When it determines that a capability under test needs to be revoked, it performs an atomic compare-and-swap of the capability to replace it with a revoked version. Only capabilities without the `VMMAP` permission are subject to revocation, so that memory allocators and other trusted, privileged software components can continue to function. By default, the revoked version of a capability is identical to the valid one, except that its tag bit is clear.

It is also possible to configure revocation to instead clear all capability permissions, so that the revoked capability is unusable but still tagged. This scheme supports applications that might benefit from identifying revoked capabilities precisely, such as language interpreters. However, it creates additional work for future revocation passes that compounds over time if the application does not manually clear capability tags, so it is not likely suitable for general-purpose applications.

5.2.2 Store-side barrier algorithm

By definition, sweeping capability revocation involves additional work that may significantly affect application performance in a number of important metrics, such as wall-clock runtime, CPU utilization, DRAM traffic, and memory footprint. However, perhaps the most immediate roadblock to practical adoption is its introduction of *stop-the-world phases*, or *pause times*, during which the application is not able to make progress. As described above, the CHERIvoke algorithm assumes that the application is paused for the entirety of a revocation sweep; such unpredictable and relatively long pauses are not acceptable for applications with interactive or real-time components. Cornucopia features two new concurrent algorithms for sweeping revocation that aim to reduce application pause times as well as wall-clock runtimes.

Any algorithm for sweeping capability revocation requires some period of time during which the running application is paused. The revoker must examine the contents of application threads' register files, which cannot be done while the threads are executing. However, the application does not have to be paused for the entire revocation pass. Sweeping revocation algorithms can thus consist of two phases: a concurrent phase that can run in parallel with the application, and a shorter stop-the-world phase that runs with the application is paused.

Concurrent algorithms for sweeping revocation must guarantee that no action taken by the application during the concurrent phase of a revocation pass can allow any capability to escape revocation. One of the novel algorithms developed for Cornucopia, inspired by past work on concurrent garbage collection [13], achieves this guarantee with *capability store-side barriers*. Capability store-side barriers use architectural support to interpose on capability stores via particular PTEs. They alert the revoker of these stores so that the application cannot bypass revocation by moving capabilities in parallel with the revocation pass.

In this subsection, I describe Cornucopia's store-side barrier algorithm. I first present a sketch of the algorithm's main invariant and high-level design. I then describe its architectural support on Morello as well as implementation details necessary for correctness. Pseudocode for the store-side barrier algorithm's revocation pass is presented in Figure 5.2 and explained further below.

Invariant The invariant for the store-side barrier algorithm is that any page that experiences a capability store will be examined by the subsequent revocation pass. If a capability store occurs concurrently with a revocation pass, then the targeted page will be examined in the current pass. This invariant guarantees that no capability stores can circumvent the revocation pass, so that when the pass is complete all capabilities that were subject to revocation are invalidated.

Design Cornucopia's store-side barrier algorithm makes use of the abstract `CapDirty` PTE bit described in the above discussion of CHERIvoke. However, rather than using the bit to indicate whether a mapping has experienced a capability store since the last time it was determined to be completely free of capabilities, as CHERIvoke does, it uses the PTE bit to indicate whether a mapping has experienced a capability store since the last time it was swept by the revoker. The corresponding fault handler sets the `CapDirty` bit, allowing future capability stores to proceed, and also triggers a store-side barrier, which alerts the revoker about the capability store.

```
1  // initial phase: concurrent
2  foreach page in application_pages
3    // check for capability observed flag
4    if !has_CO_set(page)
5      // page is known not to contain capabilities
6      continue
7
8    // execute lines 6-17 of Figure 5.1
9    revoke_page(page)
10   // clear capability stored flag
11   unset_CS(page)
12   // if no capabilities remain after the sweep of
13   // the page, clear capability observed flag
14   if !capabilities_observed(page)
15     unset_CO(page)
16
17 // scan register file and kernel stores
18 stop_application()
19
20 // final phase: world stopped
21 foreach page in application_pages
22   // check for capability stored flag
23   if !has_CS_set(page)
24     // page has not experienced capability store
25     // since the initial phase
26     continue
27
28   // execute lines 6-17 of Figure 5.1
29   revoke_page(page)
30   // clear capability stored flag
31   unset_CS(page)
32   // if no capabilities remain after the sweep of
33   // the page, clear capability observed flag
34   if !capabilities_observed(page)
35     unset_CO(page)
36
37 resume_application()
```

Figure 5.2: Pseudocode for the store-side barrier algorithm's revocation pass.

The revoker maintains two pieces of state for each page of application memory: a flag called CO for capabilities observed, which is set if a page might contain any capabilities, and a flag called CS for capabilities stored, which is set if the page has experienced a capability store since the last time CS was cleared. The flag values are managed by store-side barriers and the revoker.

Initially, empty pages are mapped with the CapDirty PTE bit and both flags clear. On a capability store that triggers a CapDirty fault, the store-side barrier sets both CO and CS for the mapped page.

Each revocation pass is split into two phases: an initial phase that runs concurrently with the application and a final stop-the-world phase during which the application is paused. During the initial, concurrent phase, the revoker makes a sweep of every page that has CO set. Pages that do not have this flag set are known not to contain any capabilities, and can safely be skipped.

To sweep each page, the revoker uses the CHERIvoke page-processing loop with CLoadTags, as presented in Figure 5.1. It invalidates all capabilities subject to revocation in the current pass. It clears the CapDirty bit for all PTEs that target the page. It clears CS, even if it encounters capabilities on the page that are not subject to revocation in the current pass. If it does not detect any valid capabilities remaining on the page, it also clears CO.

During the stop-the-world phase, after the revoker has scanned the process' register file and components of the kernel that may contain userspace capabilities, it only needs to sweep pages that have CS set. Pages with CS clear have not experienced a capability store since they were visited by the revoker in the initial phase. Even if those pages contain capabilities, the capabilities are guaranteed not to be subject to revocation in the current pass. The revoker sweeps pages and processes PTE bits and flags as described in the previous paragraph.

This algorithm design aims to reduce pause times by reducing the amount of work that the revoker must perform during the stop-the-world phase. All pages that may contain capabilities are examined while the application can continue running. Only those pages that experienced a capability store during the concurrent phase need to be examined while the application is paused: the application might have stored a capability subject to revocation after they were swept by the revoker. Care must be taken to ensure that an application running concurrently with the revoker cannot cause capabilities to escape detection in practice. I further describe the implementation details required for correctness below.

Architectural support In the Morello architecture, the functionality of the abstract CapDirty bit is implemented by concrete PTE bits that augment the baseline Armv8.2-A PTE described in Section 2.7. The SC bit, for store capability, is essentially a concrete implementation of the abstract CapDirty bit. If SC is 0, then stores of valid capabilities result in a fault. If SC is 1, then the bit has no effect.

Morello also supports hardware capability-dirty tracking, which is analogous to hardware dirty tracking as described in Section 2.7. The CDBM bit, for capability dirty bit modifier, enables hardware capability-dirty tracking. If CDBM is 0, then the bit has no effect. If CDBM is 1 and SC is 0, then capability stores proceed without causing a fault and cause hardware to set the value of SC to 1. If CDBM is 1 and SC is 1, then capability stores proceed without causing a fault.

The SC bit's value indicates whether the mapping has experienced a capability store since the last time it was cleared. Specifically, if SC is 1, then the mapping has experienced a capability store, and if SC is 0, then the mapping has not experienced a capability store. Hardware must perform an atomic compare-and-swap of the PTE in memory when attempting to set SC to ensure that the value of CDBM in memory still allows hardware capability-dirty tracking. Changes to a PTE as a result of hardware capability-dirty tracking are visible before the effects of the capability store that caused them.

Hardware capability-dirty tracking can improve the performance of sweeping capability revocation by reducing the number of faults that must be handled in software. However, it also prevents a software fault handler from executing arbitrary store-side barriers. In practice, this limitation does not affect the correctness of the algorithm. As I discuss further below, on Morello the revoker uses the presence of SC to determine when a page corresponding to the mapping has experienced a capability store, and clears SC for corresponding mappings when scanning a page.

Morello also features a concrete implementation of the abstract CLoadTags instruction. The concrete instruction is called ldct, for load capability tags. It has broadly the same semantics as CLoadTags and operates on a 64-byte region containing four capabilities. However, unlike the CLoadTags implementation suggested in the CHERIvoke paper, the implementation of ldct on the Morello SoC does not bypass caches when returning the tags for a region of uncached memory. Instead, it loads the memory's data and tags into the cache hierarchy and then returns the corresponding tags. This limits its ability to reduce DRAM traffic and cache pollution.

Implementation The primary challenge in implementing the concurrent store-side algorithm is to guarantee that no action taken by the application concurrently with the revoker's sweep can cause a capability to escape revocation. Specifically, during the concurrent phase, the application can store a capability to some page P that the revoker has already scanned. That capability could come from a page that the revoker has not yet scanned, and the capability might be subject to revocation in the current pass. This store must be detected, and the revoker must scan P during the final phase, or the capability stored there would escape revocation. Enforcing this property is made more complex by a number of factors:

- PTE bits to track capability stores apply to particular virtual-physical mappings, and a single physical page of memory can be the target of many such mappings.
- Hardware capability-dirty tracking, along with other hardware-managed PTE bits, means that PTEs might be updated by MMU hardware concurrently with code running on a CPU core, even on a single-core system. As a result, the revoker cannot rely on software locking to synchronize PTE-modifying operations. It must use atomic operations and memory fences when modifying PTEs to ensure that PTE updates are observed and ordered correctly relative to other operations.
- On a multicore system like Morello, each CPU core has its own TLB whose cached PTEs are not coherent with those of other TLBs or the underlying values in memory. This means that, when the revoker clears the SC bit of a PTE in memory while scanning a page, it needs to invalidate potentially stale versions of that PTE cached in TLBs. Otherwise, if a CPU core stores a capability via a stale version of the PTE

cached in its TLB that already has SC set, the store will not cause SC to be set in the in-memory version of the PTE. In other words, the capability store would not be tracked, and the stored capability might escape revocation. To avoid this situation, the revoker must perform a *TLB shootdown* for the PTE, an expensive synchronous operation, usually involving inter-processor interrupts, that flushes an entry in every core's TLB. A *global TLB shootdown* is a TLB shootdown that flushes the entire TLB for each core.

- Integrating with the rest of the kernel adds significant incidental complexity to the revoker's implementation. For example, the revoker must correctly modify and use complex interfaces that govern MMU mappings, virtual memory abstractions, and system calls that might interfere with revocation.

To address these factors as efficiently as possible, the algorithm's implementation aims to minimize synchronization overhead. It treats PTEs as caches that lazily propagate capability-dirty information in batches to the per-page metadata flags CS and CO. Each virtual page could have its own values for CS and CO. In practice, however, the revoker stores these flags in aggregate for each physical page; doing so works naturally with existing kernel data structures and also allows the revoker to avoid scanning the same physical page twice in some scenarios.

The revoker starts each phase of a revocation pass, i.e. both the concurrent and stop-the-world phases, by iterating over the entire virtual address space and propagating the state of PTEs to per-page metadata. For each PTE, it checks the SC bit; if the bit is set, then the revoker sets CS and CO on the page of memory targeted by the mapping. The revoker then atomically clears the SC bit in the in-memory copy of the PTE. Once all PTEs have been processed, the revoker performs a single global TLB shootdown and executes a fence to synchronize system state. Batching in this way allows the all capability-dirty propagation to share a single TLB shootdown and acquisition of kernel structure locks.

After PTE information has been propagated to page metadata, the revoker proceeds by performing another sweep of the virtual address space. In the concurrent phase, it visits all pages that have CO set, and in the stop-the-world phase, it only needs to visit pages that have CS set. After visiting a page, it clears CS and if it does not encounter any non-revoked capabilities on the page, it clears CO. It does not modify the SC bit for any PTEs at this point; it already did so when propagating their state to the page metadata.

The application might store capabilities via any PTE at any point during the concurrent phase. All such cases must be analyzed to confirm that they do not allow any capabilities to escape revocation:

- If the capability store occurs before the revoker checks the value of the PTE during the batched propagation of PTE state to page metadata, then it will be reflected in the in-memory copy of the PTE observed by the revoker during state propagation.

- If the capability store occurs after the revoker checks the value of the PTE during batched state propagation, but before the global TLB shootdown, then there are two possibilities.

  If the store occurs via a cached version of the PTE that did not have SC set, then it causes the hardware capability-dirty tracking mechanism to set SC in the in-memory copy of the PTE, but this information does not get propagated to page metadata

for the concurrent phase. However, because SC has been set in memory, the PTE will be processed during the stop-the-world phase and so the written capability will not escape revocation.

If the store occurs via a cached version of the PTE that did have SC set, then the hardware capability-dirty tracking mechanism does not set SC in the in-memory copy of the PTE. However, the presence of a cached version of the PTE with SC set at this time implies that the PTE experienced a capability-dirtying store since the last revocation pass. If that capability-dirtying store occurred before the revoker checked the value of the PTE during batched state propagation, then the page will be scanned during the incremental pass and the written capability will not escape revocation. If that capability-dirtying store occurred after the revoker checked the value of the PTE during batched state propagation, then SC will have been set in memory and the PTE will be processed during the stop-the-world phase, so the written capability will not escape revocation.

- If the capability store occurs after the global TLB shootdown, then it will be reflected in the in-memory copy of the PTE observed by the revoker during the stop-the-world phase.

- It is also possible for the application to disrupt the revoker's activity by storing to the location of a capability between when the revoker loads it for testing in the shadow bitmap and when the revoker performs a compare-and-swap to revoke it. In this case, the revoker's compare-and-swap operation will fail, and the revoker will set CS on the page to ensure it is visited again during the stop-the-world phase.

This case analysis of the concurrent phase demonstrates that it is not possible for capabilities to escape revocation by the store-side barrier algorithm. The stop-the-world phase of the algorithm is not subject to concurrent application execution. It scans kernel subsystems that might hold application capabilities in addition to application register files and pages of memory with CS set. The revocation pass is then complete, and the application can resume execution.

Discussion The store-side barrier algorithm aims to reduce pause times by scanning most virtual pages of application memory concurrently with application execution. It is reasonable to expect that an application will store capabilities to significantly more pages between revocation passes than during the concurrent phase of a particular pass, so the number of pages that must be scanned with the application paused should be less than for the CHERIvoke baseline. However, virtual pages can end up being scanned twice in a single revocation pass: once during the concurrent phase, and, if they experienced a capability store in the interim, again during the stop-the-world phase. This characteristic of the algorithm might result in significant duplicated work, especially in the presence of pathological application behavior. Additionally, the store-side barrier algorithm still requires iterating over the entire virtual address space with the application paused to propagate PTE state to per-page metadata. With a different algorithm design and novel architectural features, it is possible to improve some of the negative performance characteristics of the store-side barrier algorithm.

5.2.3 Load-side barrier algorithm

```
1  // scan register file and kernel stores
2  stop_application()
3
4  foreach page in application_pages
5    // clear CapLoad to prevent capability loads
6    unset_CapLoad(page)
7
8  // handlers execute revoke_page()
9  // asynchronously to sweep pages as they
10 // experience faulting capability loads
11 resume_application()
12
13 foreach page in application_pages
14   // execute lines 6-17 of Figure 5.1
15   revoke_page(page)
16   // set CapLoad to allow capability loads
17   set_CapLoad(page)
```

Figure 5.3: Pseudocode for the load-side barrier algorithm's revocation pass.

Another technique for concurrent sweeping capability revocation, also inspired by the garbage collection literature [112], involves the use of *capability load-side barriers*. While store-side barriers track capability flow by interposing on capability stores, load-side barriers do so by interposing on capability loads. Relative to the store-side algorithm, Cornucopia's load-side barrier algorithm requires additional architectural support, but it reduces duplicated work and pause times. In this subsection, I describe the main invariant, design, architectural support, and implementation details for the load-side algorithm. Pseudocode for the load-side barrier algorithm's revocation pass is presented in Figure 5.3 and explained further below.

Invariant The invariant for the load-side barrier algorithm is that, once a revocation pass has been initiated, no register can load a valid capability until the page containing it has been swept by the revoker. This invariant guarantees that capabilities subject to revocation are invalidated before they can be loaded and used.

Design For the purposes of understanding the algorithm's design, consider an abstract PTE bit `CapLoad`. When `CapLoad` is 0, capability loads from the corresponding page result in a fault, and when `CapLoad` is 1, the bit has no effect. As I describe below, the novel architectural features on Morello to support load-side barriers are more sophisticated than this abstract bit, but `CapLoad` suffices to explain the algorithm at a high level.

Sweeping revocation with load-side barriers differs significantly from the store-side barrier algorithm. Store-side barriers facilitate an initial phase that runs concurrently with the application. A final phase then examines register files and kernel subsystems, and it performs another scan of memory to catch any capabilities that were stored by the

application during the initial phase, because those capabilities might have been subject to revocation but not yet examined by the revoker.

On the other hand, the load-side barrier algorithm begins with a stop-the-world phase. While the application is paused, the revoker examines thread register files and kernel subsystems, and it sets `CapLoad` to 0 for every PTE in the application's virtual address space. Then, the concurrent phase begins and the application is allowed to resume execution. In this phase, the revoker sweeps every page in an application's virtual address space and revokes all capabilities that are subject to revocation in the current pass, using the same page-processing loop presented in Figure 5.1. Once it has finished processing a page, it sets `CapLoad` to 1 so that capability loads can proceed without faulting. If the application attempts to load a capability from a page before the revoker has processed it, the load results in a fault. The fault handler calls the revoker's page-processing loop on the targeted page, and once the page has been scanned and its capabilities revoked as appropriate, it sets `CapLoad` to 1 and the load can proceed. Once the revoker finishes scanning all of memory, the revocation pass is complete.

The load-side barrier algorithm guarantees that, during the concurrent phase of a revocation pass, the application's register files and kernel subsystems do not hold any valid capabilities that are subject to revocation in the pass. The application is also unable to load any valid capabilities that are not subject to revocation, because it can only load capabilities from a page after the revoker has scanned it. The application is thus unable to store capabilities in a way that would allow any capability to escape revocation, and no interposition on capability stores is required.

Architectural support While the abstract `CapLoad` bit facilitates a correct implementation of sweeping revocation with load-side barriers, such an implementation would require iterating over the application's entire virtual address space while it is paused to set `CapLoad` to 0 for all PTEs. Here I introduce a novel, more sophisticated scheme for faulting on capability loads that obviates the need for this iteration. In a *capability load generation* scheme, each PTE has a one-bit local capability load generation counter, and the architecture also supports a single global capability load generation counter, controllable only by the kernel, that is associated with a particular application's virtual address space. When an application attempts to load a capability via some PTE, the load will fault if value of the PTE's local counter does not match that of the global counter. With such a capability load generation scheme supported by the architecture, the stop-the-world phase of the load-side barrier algorithm can simply toggle the global capability load generation counter instead of modifying each PTE in the application's virtual address space.

The Morello architecture added capability load generation counters to support this work on sweeping capability revocation. Each PTE has bits `LC[2:1]`, for load capability, and when `LC[2]` is 1, `LC[1]` serves as the PTE's local capability load generation counter. Bit `TGEN0` of the platform capability control register `CCTLR_EL1` serves as the global capability load generation counter. In the following text, I will refer to `LC[1]` as `LCLG`, for local capability load generation, and `CCTLR_EL1.TGEN0` as `GCLG`, for global capability load generation.

These capability load generation counters apply to capability loads of userspace memory from both the privileged kernel context and the non-privileged application context. To avoid causing faults on capability loads while scanning pages for revocation, the revoker uses the kernel's privileged direct-mapped region to access userspace memory.

Implementation Implementing concurrent sweeping revocation with capability load generations requires ensuring that, during a revocation pass, no capabilities can be loaded via a PTE until the corresponding page of memory has been scanned by the revoker. Before a revocation pass starts, each PTE's `LCLG` bit is equal to `GCLG`, because the previous revocation pass is not complete until it has scanned every virtual page of application memory. During the stop-the-world phase that starts a revocation pass, the revoker scans application register files and kernel subsystems, toggles `GCLG`, performs a global TLB shootdown, and executes a fence to synchronize system state.

A global TLB shootdown must take place between the end of one revocation pass and the end of the stop-the-world phase of the subsequent pass. To see why this is necessary, consider a revocation pass that toggles `GCLG` from 0 to 1 during its stop-the-world phase. It then changes `LCLG` for each PTE from 0 to 1 during the concurrent phase, but stale copies of some PTEs with `LCLG` 0 persist in TLB entries for some cores. During the next pass, the revoker toggles `GCLG` from 1 to 0, and after the stop-the-world phase, the stale TLB entries can be used to load capabilities without triggering a load-side barrier. Performing a global TLB shootdown between the end of one revocation pass and the end of the stop-the-world phase of the next prevents this type of bypass from occurring. Doing so at the end of the stop-the-world phase is natural, because resuming the application and performing the shootdown both require expensive inter-processor interrupts, and it may be possible to batch them.

After the initial stop-the-world phase, the revoker scans the application's entire virtual address space concurrently with the application. For each virtual page, it checks whether the PTE's `LCLG` bit in memory is equal to the value of `GCLG`. If so, the page has already been scanned by the fault handler, and so it can be skipped. Otherwise, the revoker scans the page for capabilities to revoke, performs a fence operation, then sets the page's `LCLG` PTE bit in memory to that of `GCLG`.

If the application attempts to load a capability during the concurrent phase, and the load results in a capability load generation fault, then the fault handler first checks to see whether the `LCLG` bit in the targeted virtual page's in-memory PTE is equal to the value of `GCLG`. In that case, the page has already been swept, either by the revoker or by a fault handler on a different core, and the access is being performed via a stale TLB entry cached by the current core; the fault handler can simply reload the TLB entry and allow the load to proceed. Otherwise, the fault handler scans the page for capabilities to revoke, performs a memory fence operation, then sets the page's `LCLG` PTE bit in memory to that of `GCLG`.

The revocation pass is complete, and another can be initiated, once the revoker has scanned the application's entire virtual address space in the concurrent phase. This algorithm is designed to scan each virtual page one time per revocation pass. The only way a page can be scanned more than once in a revocation pass is if a capability load generation fault handler and the revoker process a page in parallel, and one starts after another but before it has updated the PTE's `LCLG` bit. This scenario does not affect correctness and is unlikely to occur in practice.

Discussion Relative to the store-side barrier algorithm, the load-side algorithm significantly reduces the amount of work that must be performed with the application paused. Rather than iterating over all application PTEs during the stop-the-world phase, it simply toggles the value of the process-wide `GCLG` bit. It also requires fewer global TLB shoot-

downs per revocation pass, and it nearly eliminates the possibility of scanning a virtual page twice in the same pass.

On the other hand, the store-side algorithm can skip pages that are known not to contain capabilities in a straightforward way, and it only relies on architectural features that track capability stores. The load-side algorithm as presented must scan every page in the application's virtual address space, even those that do not contain capabilities, during the concurrent phase. Depending on application characteristics, this limitation may result in a significant CPU utilization cost. The implementation of the load-side algorithm in CheriBSD also employs a form of capability-dirty tracking, so that the revoker can skip scanning pages that are known not to contain any capabilities, which reduces this cost.

5.3 Temporally safe heap allocators

In this section, I discuss how userspace allocators can use Cornucopia to guarantee heap temporal safety. I first systematize a list of prerequisites, inspired by security weaknesses I discovered while investigating properties of CheriABI's system heap allocator, that a memory allocator must meet before it can be made temporally safe. I then describe the changes that must be made to an allocator meeting these prerequisites in order for it to interface with Cornucopia directly. Finally, I introduce a generic wrapper module that allows prerequisite-meeting allocators to leverage Cornucopia with very minor source-code changes. This wrapper module demonstrates the surprising result that logic to enforce temporal safety with Cornucopia can be encapsulated into an allocator-independent library.

5.3.1 Heap allocator prerequisites

In order for a memory allocator to facilitate temporal safety with Cornucopia, it must meet the following prerequisites in addition to supporting pure-capability compilation:

- The allocator must clear the `VMMAP` permission on all capabilities returned to an application. Cornucopia only targets capabilities without this permission, so capabilities with `VMMAP` set held by the application could be used to bypass revocation. The permission must also be cleared to ensure that the application cannot modify parts of the address space used by the heap allocator via the `mmap()` family of system calls, which check for the presence of `VMMAP` on capability inputs from CheriABI processes.

- The allocator must zero heap memory before it is allocated to an application. Otherwise, uninitialized reads can result in a form of heap memory object aliasing, even with Cornucopia revocation enabled.

- The allocator must set fine-grained bounds on capabilities to heap memory objects that are returned to the application. Otherwise, spatial safety violations on the heap could allow the field of some heap memory object to be corrupted by an out-of-bounds write in a different object, even if the two objects do not directly alias the same region of heap memory. Furthermore, fine-grained bounds prevent the application from accessing heap-internal metadata stored just outside of an allocated heap memory object, which is a concern for some allocator designs.

- `realloc()` must be implemented in terms of `malloc()` and `free()`. More specifically, the `realloc()` implementation must first allocate a new heap region of the requested size, perform a memory copy, and then free the old region. It cannot perform optimizations that grow or, perhaps more importantly for temporal safety, shrink existing regions of the heap. If it did, valid capabilities to old, pre-`realloc()` heap memory objects could persist after the `realloc()` call without being subject to revocation.

- The allocator must be resilient in the face of attacker-controlled inputs to heap memory allocation API functions. Otherwise, it may be possible for an attacker to corrupt allocator-internal state and violate desired security properties.

This last point requires some additional explanation. Recall from Section 3.3.2 that, in the threat model for CheriABI, the attacker is assumed to have either control over an application's input stream or arbitrary code execution in an unprivileged application DSO. Applying this threat model in the context of memory allocators, the attacker is assumed to be able to invoke `malloc()`, `free()`, and other memory allocation API functions arbitrarily, and with arbitrary arguments. The attacker can utilize only valid capabilities that have been made available to them, but they can perform arbitrary loads and stores via those capabilities. This threat model corresponds to real-world situations of interest, such as an adversary-owned webpage that can control memory allocations in a targeted JavaScript interpreter. A similar model applies to an adversary-controlled process using system calls to influence the heap of a target kernel, but kernel heap temporal safety is out of scope for the present discussion.

With the above threat model, if an allocator is not hardened against malformed inputs to its API functions, an attacker may be able to corrupt its internal state and violate security properties. For example, CheriBSD uses a version of the `jemalloc` allocator that returns bounded capabilities to heap memory objects for CheriABI processes [32]. While analyzing the security of this memory allocator in 2019, I discovered that, despite correctly setting bounds on heap allocations, it did not effectively validate capabilities passed to `free()` [77].

If a consumer made a call to `malloc()` for an allocation of size L, it would receive a capability with a length of L whose base and address were both equal to some address B on the heap. If the consumer then incremented the capability's address by some value c, to $B + c$, and passed the capability to `free()`, the modified capability would be saved onto an allocator-internal list of regions to be re-allocated. A subsequent `malloc()` request of size L would cause the allocator to pop the modified capability off of the list and refine its base and bounds before returning it to the application. The capability refined and returned by the allocator would have a length of L, but a base and address of $B + c$. This capability would allow the consumer to access c bytes into the adjacent heap memory object located at $B + L$, thereby violating heap spatial safety.

This discovery prompted me to systematize the capability-related validation that must be performed in the implementations of `free()`, `realloc()`, and other heap memory allocation API functions that accept capability inputs. Some of these requirements were identified simultaneously by others after my initial bug report. A researcher at the Microsoft Security Response Center reported a bug in CheriBSD's `jemalloc` related to omitted validation of capability tags for inputs to `free()` [55]; Laurie Tratt of King's College London [94] and a participant in a CHERI red-teaming exercise [109] identified

bugs related to the `realloc()` function.

The validation steps that must be performed on capability inputs to memory allocation API functions to make allocators resilient in the face of capability-aware attackers are as follows:

- Capability inputs must be tagged, have in-bounds addresses, and have appropriate permissions. Otherwise, an adversary might be able to call `free()` on objects they do not own or forge capabilities to memory they do not own. Appropriate permissions are a subset of the permissions returned by `malloc()`.
- Capability inputs must have addresses that point to the beginning of a non-free region of memory that was previously returned by `malloc()`. Otherwise, an attacker may be able to corrupt the allocator's internal state by freeing capabilities with modified bases or addresses, as described above.

Modifying memory allocation API functions to check capability tags, permissions, and bounds is straightforward. Techniques for validating that a capability's address points to the beginning of an allocated region of memory depend on the allocator's implementation. For slab allocators, like `jemalloc` [42] and Microsoft's `snmalloc` [66], the allocator can verify that the address is within an existing slab and at a position consistent with the beginning of an allocation for the slab's size-class. Boundary tag allocators like `dlmalloc` [63], which mixes object sizes within an arena and stores metadata for an allocation just outside of its bounds, can validate the capability's address by checking for unforgeable capabilities within this metadata.

5.3.2 Adapting heap allocators

A few modifications are necessary to adapt allocators that meet the above prerequisites into temporally safe allocators that use Cornucopia. When a heap memory object is passed to `free()`, the allocator must not make it immediately available for reuse but instead place it in a quarantine buffer, as described above. Additionally, metadata to track which regions of memory are in the quarantine buffer cannot be stored in the regions themselves, because the application can retain and use valid capabilities to those regions until after a revocation pass.

The allocator must also keep track of the quarantine buffer's size. Once it passes some threshold, either a fixed number or a proportion of the total heap size, the allocator must invoke the `cheri_revoke()` system call to trigger a revocation pass. Regions of the shadow bitmap corresponding to quarantined memory must be set before the system call and cleared afterwards. Once Cornucopia has completed the revocation pass, memory in the quarantine buffer can safely be reallocated.

These modifications to support Cornucopia can be implemented concretely in a number of different ways. Key design decisions include the quarantine buffer size threshold; when the quarantine buffer's size is checked to initiate a revocation pass; the scheme used to track memory regions in the quarantine buffer; and the threading context in which revocation-related work, such as interacting with the shadow bitmap, calling `cheri_revoke()`, and processing the quarantine buffer, is performed. An allocator's design, as well as the constraints of the system running the application, determine the optimal choices.

The number of design choices for allocator integration highlight Cornucopia's flexibility. However, while these modifications are straightforward to enumerate and describe, implementing them in practice can require a significant amount of developer effort. Contemporary memory allocators are complex codebases, and tightly integrating them with Cornucopia correctly involves significant source-code changes, as I demonstrate in Section 5.4.3. A lower-effort solution that augments existing allocators with temporal safety while respecting their allocation policies would prevent this hurdle from becoming a roadblock to Cornucopia adoption.

5.3.3 Allocator-independent wrapper

In this section, I describe the design of a generic wrapper module that can augment memory allocators with temporal safety. Targeted allocators only need to meet the prerequisites of Section 5.3.1. The wrapper module's design choices may not be optimal for particular wrapped allocators, and it incurs some additional costs relative to direct integration with Cornucopia, but it allows for the provision of temporal safety with very little source-code change required in the allocator. This is useful for contemporary systems, which utilize a large number of heap memory allocators whose complex codebases might require significant developer effort to integrate with Cornucopia directly. It also demonstrates that the logic required to enforce temporal safety with Cornucopia can be implemented generically, without direct access to allocator internals, which was an unexpected result.

The wrapper is a dynamic shared object (DSO) that interposes on calls to memory allocation functions such as `malloc()` and `free()`. When the wrapper DSO is preloaded by the runtime linker, its exported functions perform the operations and bookkeeping necessary to provide temporal safety with Cornucopia; they use the corresponding functions exported by the wrapped heap allocator to actually manage allocations. For example, with the wrapper preloaded, a call to `malloc()` made by the application will resolve to the wrapper's `malloc()` implementation, which will subsequently make a call to the wrapped allocator's `malloc()` implementation. The wrapper also interposes on calls that the wrapped allocator makes to `mmap()`. The wrapper calls the `mmap()` system call, passes the returned capability to `cheri_revoke_shadow()` to gain access to the corresponding region in the shadow bitmap, and then returns the capability to the wrapped allocator. A schematic diagram of how the wrapper interacts with other system components is presented in Figure 5.4.

The only additional demand the wrapper makes on wrapped heap allocators is that they export a function called `malloc_wrapper_underlying_allocation()`. This function takes in a capability referencing some heap allocation, validates that it is not malformed as described in Section 5.3.1, and returns a capability whose bounds match those of the capability returned by the call to `malloc()` for the allocation. The returned capability also has the `VMMAP` permission set, making it privileged. Because of this, `malloc_wrapper_underlying_allocation()` should only be exported to the wrapper and not made available as part of the allocator's public API.

Information from `malloc_wrapper_underlying_allocation()` allows the wrapper to track the quarantine size correctly and to revoke references to any part of the original allocation; the bounds of the capability passed to the wrapper's `free()` function could have been shrunk by the application. It also allows the wrapper to use capabilities with the `VMMAP` permission in its internal bookkeeping. These capabilities will not be

invalidated during revocation passes, so they will later be accepted by the wrapped allocator's implementation of `free()`.

Implementing `malloc_wrapper_underlying_allocation()` is straightforward. Many memory allocators already export a standards-extending function called `malloc_usable_size()`, which accepts a pointer and returns the usable size of the underlying allocation that it points to. The implementation of `malloc_wrapper_underlying_allocation()` is nearly identical, except it returns a privileged capability to the underlying allocation. As such, adding support for `malloc_wrapper_underlying_allocation()` typically requires very small modifications to source code.

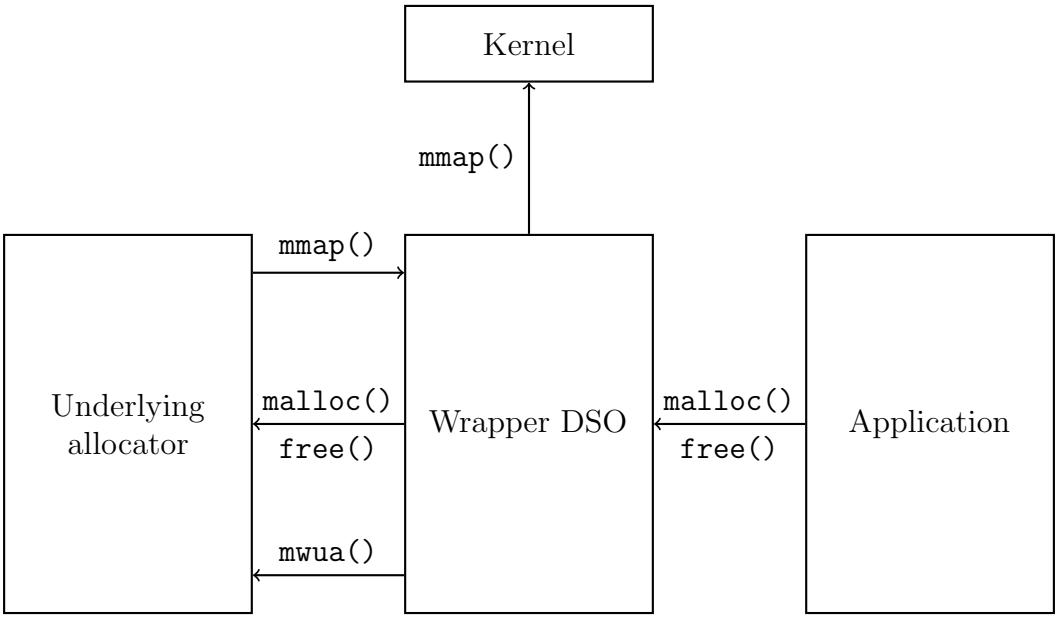


Figure 5.4: Schematic diagram of how the allocator-independent wrapper interacts with other system components. Arrows represent function calls. Arrows labeled with `malloc()` and `free()` signify all functions in the `malloc()` interface. `mwua()` signifies `malloc_wrapper_underlying_allocation()`.

Here I describe the operation of the wrapper's `malloc()` and `free()` functions. Other allocation and deallocation functions are similar. The wrapper's `malloc()` calls the wrapped allocator's `malloc()`, increments a counter that tracks the total heap size by the length of the returned capability, and returns the capability. The wrapper's `free()` function:

- Validates the passed-in capability using `malloc_wrapper_underlying_allocation()`.
- Atomically checks and paints the corresponding region of the shadow bitmap so that a region of memory cannot be added to the quarantine buffer twice, even by concurrently executing threads.
- Increments a counter that tracks the quarantine buffer size by the length of the capability returned by `malloc_wrapper_underlying_allocation()`.
- Adds the region corresponding to the capability returned from `malloc_wrapper_underlying_allocation()` to the quarantine buffer.

The quarantine buffer is implemented as a linked list of slabs that contain descriptors for quarantined memory regions, and the descriptors are simply the privileged capabilities returned by `malloc_wrapper_underlying_allocation()`. The wrapper allocates memory for these slabs using the `mmap()` system call.

By default, the wrapper checks whether to initiate a revocation pass on calls to `malloc()`. This prevents spurious revocation passes from taking place as an application frees significant amounts of memory before exiting. However, the wrapper can also be configured to initiate revocation passes on calls to `free()`.

The quarantine buffer threshold size is configurable as either a fixed value or a proportion of the total heap size. When the thresholds are met, the wrapper:

- Calls the `cheri_revoke()` system call to initiate a revocation pass. The sweeping revocation algorithm used is configurable.
- Subtracts the quarantine size counter from the heap size counter and resets the quarantine size counter to zero.
- Iterates through its slabs of quarantined region descriptors, clearing the corresponding bits in the shadow bitmap and freeing the quarantined regions back to the wrapped allocator.

The wrapper can be configured to perform the above work synchronously in an application thread or to spawn an offload thread that performs the work asynchronously. The offload thread allows single-threaded allocators and applications to benefit from concurrent revocation algorithms; without some kind of offload thread, the `cheri_revoke()` system call would consume the single application thread even if it used using a concurrent algorithm for sweeping revocation. When configured to use an offload thread, the wrapper minimizes revocation-related work that must be performed in the application thread. Specifically, the application thread adds memory regions to the quarantine buffer without any validation, and when the revocation threshold is reached, passes the quarantine buffer to the offload thread. The offload thread then performs all validation, shadow bitmap management, and freeing in addition to making the `cheri_revoke()` system call. While the offload thread processes a quarantine buffer, the application thread can continue working and filling up a second one. The application thread blocks if this second quarantine buffer passes the revocation threshold while asynchronous revocation is still in progress.

Using this generic wrapper comes with some performance costs relative to direct integration with an allocator. One of these costs is that the design choices made in the wrapper may not be optimal for the wrapped allocator. Another is that operations to validate and re-derive a more privileged version of each freed capability must be performed twice: once when the wrapper calls `malloc_wrapper_underlying_allocation()` and once when the capability is freed back to the wrapped allocator. In an integrated design, the allocator might avoid this duplicated work. A further cost is that data structures for storing the quarantine buffer must be maintained externally to the allocator, whereas an integrated design might reuse existing allocator metadata to track the quarantine list.

5.4 Evaluation

I now evaluate Cornucopia and the generic userspace wrapper module according to the hypotheses set out at the beginning of this chapter. More specifically, I consider whether

Cornucopia-augmented allocators can effectively guarantee userspace heap temporal safety in practice on CHERI hardware, whether the novel algorithms implemented in Cornucopia improve performance relative to the CHERIvoke baseline, and whether the performance cost of the wrapper module relative to direct integration is worth its source-code compatibility benefits. I carried out my experiments using Arm's Morello SoC [71], the first industrial-quality board to incorporate CHERI features. Morello was released in early 2022, and these experiments contain the first published performance results from the platform.

As discussed in Section 1.3, I made significant contributions to enable performance measurement on the Morello SoC. However, the platform currently has a number of limitations due to its immaturity, which I described in detail in Section 2.4.1. These limitations make it challenging to measure the performance overhead of pure-capability code relative to normal 64-bit Arm code as well as the performance overhead of adding new code paths to existing pure-capability programs.

The experiments I describe below are formulated to work around these challenges. In particular, I compare the performance of new sweeping revocation algorithms relative to the CHERIvoke algorithm as a baseline, rather than using a CheriABI binary without heap temporal safety or a normal 64-bit Arm binary as a baseline. The experiments are sufficient to evaluate the hypotheses presented in this chapter; however, further exploration of the performance cost of sweeping revocation relative to these other two baselines should be carried out once the platform limitations have been addressed.

Platform configuration The Morello SoC was configured to store capability tags in RAM error-correcting-code bits. Benchmarks were run on CheriBSD with a CheriABI userspace. The kernel was a normal Armv8-A.2 binary augmented with *hybrid* capability support to facilitate the CheriABI process environment. Kernel configuration and build-time options were set to disable debugging features that negatively affect performance.

The userspace software artefacts under test included three memory allocators adapted to support pure-capability compilation: the classic boundary-tag allocator `dlmalloc` [63]; the slab allocator `jemalloc` [42], which is the default system allocator in FreeBSD; and Microsoft's relatively new `snmalloc` [66] slab allocator. Allocators were evaluated both with direct Cornucopia integration and with the use of the generic wrapper module.

Test workloads came from a subset of the SPEC CPU 2006 benchmark suite [108]. I chose to use the SPEC suite because it is well characterized and has been used in in previous work on heap memory allocator performance. The suite also figures multiple workload configurations for each benchmark, including short-running *test* configurations and long-running *ref* configurations. The *test* configurations were particularly useful while developing the designs under test on FPGA- and software-based models that are significantly less powerful than the Morello SoC. Eight of the twelve SPECint benchmarks build and run successfully as pure-capability code: `astar`, `bzip2`, `gobmk`, `hmmer`, `libquantum`, `omnetpp`, `sjeng`, and `xalancbmk`. All userspace code was compiled into dynamically linked CheriABI binaries. Threads for the benchmarks under test were pinned to particular cores and all other system work was pinned to a disjoint set of cores.

5.4.1 Effectiveness

To evaluate the effectiveness of heap temporal safety with sweeping revocation on real CHERI hardware, I used the Juliet test suite [15, 59]. The Juliet suite is a large corpus of

C/C++ programs with known vulnerabilities that is designed to assess the effectiveness of static analyzers and other software-assurance tools. Each test in the suite is a program in source form, and the tests are classified according to the type of vulnerability that they concern. The tests are synthetic, i.e. they have been generated to exercise a single well-characterized vulnerability with a much simpler structure than vulnerable production code that might have inspired the case. This characteristic is useful for CHERI; simple test programs can be compiled as pure-capability code without modification.

For each test in the Juliet suite, I built two CheriABI executables. One included the known vulnerability, and the other did not. In the below discussion, I refer to the executable containing the vulnerability as the bad executable, and the vulnerability-free executable as the good executable.

The class of test cases in the Juliet suite most directly relevant to heap temporal safety is Use After Free, in which the vulnerability-containing cases attempt to access heap memory via a stale, i.e. previously freed, pointer. However, the Juliet suite is designed to test tools like static analyzers and sanitizers that actively identify all scenarios in which a region of heap memory is accessed after it has been freed. When configured with a non-zero quarantine buffer, userspace heap allocators that utilize Cornucopia silently allow non-exploitable uses after free to take place. The Juliet test cases do not free enough memory to fill a meaningfully sized quarantine buffer and cause a revocation pass. As a result, even though Cornucopia-based protection with a quarantine buffer would mitigate the vulnerabilities by preventing the reallocation of freed memory, that mitigation would not manifest as an alert or exception that could be detected by a test runner.

In order to make Cornucopia-based temporal safety compatible with the Juliet tests, I built versions of the generic wrapper module for each sweeping revocation algorithm that was tuned to use a quarantine buffer size of zero. This causes the wrapper to operate like a sanitizer, performing a revocation pass on every call to `free()`. All attempts to access heap memory via a freed pointer then result in a CHERI exception.

I tested two allocator configurations: one was CheriBSD's spatially safe default system allocator `jemalloc`, and the other was `jemalloc` wrapped in the generic temporal-safety modules built with a quarantine buffer size of zero. For all 393 test cases in the Use After Free class, the good executable exited successfully using both the unwrapped and wrapped versions of `jemalloc`. The bad executable exited successfully with the non-wrapped version of `jemalloc`, indicating that the use after free was allowed to occur. It resulted in a CHERI exception with the wrapped versions, indicating that the use after free was mitigated.

The ability of Cornucopia-based temporal safety to mitigate all use-after-free cases when configured with a quarantine buffer size of zero suggests that, with a non-zero quarantine buffer, it would successfully mitigate use-after-reallocation scenarios. Freed memory stored in the quarantine buffer would still be accessible via stale pointers, but a revocation pass is guaranteed to take place before the memory is reallocated, so any attempts to access reallocated memory via a stale pointer would result in a CHERI exception. Furthermore, while the Juliet test cases are synthetic, real-world temporal safety vulnerabilities, such as those described in the analysis of iOS exploit chains in Section 3.4.1, are not qualitatively different. These results suggest that it is possible to implement sweeping capability revocation on real CHERI hardware to mitigate heap temporal safety vulnerabilities in practice.

5.4.2 Sweeping revocation algorithms

In this subsection, I evaluate to what extent concurrent sweeping revocation algorithms using store-side and load-side barriers improve performance relative to the CHERIvoke baseline. SPEC benchmarks were run with *ref* workloads using a version of `snmalloc` that was adapted to support CheriABI and wrapped with the generic Cornucopia module. The sweeping revocation algorithm used varied across the CHERIvoke, store-side, and load-side algorithms. While the benchmarks are all single-threaded, experiments were run both with and without the wrapper's offload thread enabled to study the behaviors of the different algorithms in those cases. All other aspects of execution were held constant.

The wrapper's quarantine threshold was set to a fixed value of 20 MiB. A fixed threshold was used, rather than a ratio of the total heap size, to ensure fair comparability across sweeping revocation methods. In the offload-thread case, the application might continue filling a second quarantine buffer, and in particular make allocations that increase the heap size, concurrently with the revoker's operation. Because revocation passes take different amounts of time depending on the algorithm used, heap sizes across the different algorithms then diverge during execution. With the quarantine threshold as a ratio of heap size, the number of times the threshold is hit, and thus the number of revocation passes, also diverges. Comparisons between the algorithms are then unfair, because, as stated before, the performance cost of sweeping revocation is a trade-off between memory use and computational overhead: the more memory used to store the quarantine buffer, and the fewer revocation passes, the less computational overhead there will be. Using a fixed threshold value does not completely eliminate this effect, but it prevents divergence in the number of revocation passes so that the different algorithms can be compared fairly.

Overview Figure 5.5 provides an overview of the performance results for running all supported SPEC benchmarks, with an offload thread, across the different sweeping revocation algorithms. Each horizontal axis in the chart measures a different metric, and each vertical column of bar clusters corresponds to a particular benchmark. The benchmark names appear across the bottom of the chart. Within each cluster, the blue and left-most bar represents the metric's value for the CHERIvoke algorithm, the orange and center bar for the store-side barrier algorithm, and the green and right-most bar for the load-side barrier algorithm. Numerical values appear under each bar.

As a whole, these data show that, with additional architectural support from Morello, the load-side barrier algorithm achieves extremely reduced pause times and wall-clock runtimes for applications augmented with temporally safe heaps. Its pause times are also constant across application workloads. The store-side algorithm, which uses the same architectural features as the CHERIvoke baseline, also achieves reduced pause times and wall-clock runtimes, but they are not as low as those of the load-side algorithm, and pause times scale with the application's maximum RSS.

The top axis measures, in milliseconds, the length of the longest span of time that the application spends paused as a result of sweeping revocation. As discussed previously, the length of a revocation pass and its corresponding pause time can depend on how many of the application's pages of memory contain capabilities. The number of capability-containing pages varies during execution and tends to increase over time. To account for this effect, the axis presents the worst-case pause time across all of the revocation passes. These measurements refer only to pause times during which the kernel prevents the application from doing work; times during which the application has filled up two quarantine buffers

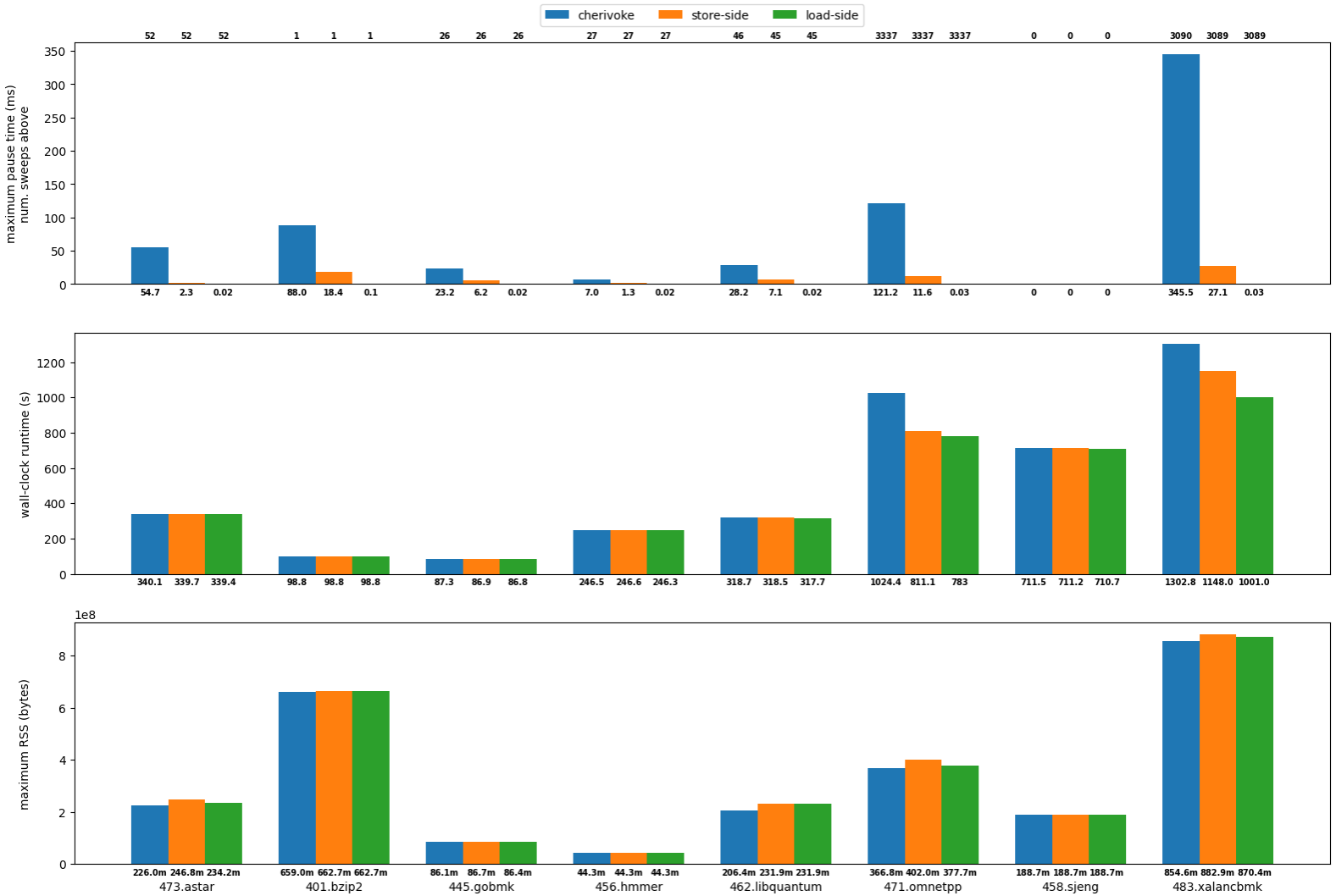


Figure 5.5: Overview of the performance of sweeping revocation algorithms for SPEC benchmarks with an offload thread. Lower is better for all metrics. With the use of new architectural features, the load-side barrier algorithm achieves extreme pause-time and wall-clock runtime reductions.

and the heap allocator chooses to wait while the kernel finishes processing one of them are not counted.

As might be expected, the CHERIvoke algorithm has the longest pause times across all of the benchmarks, because the application must remain paused for the entirety of the revocation pass. In the `xalancbmk` case, the application experiences pauses of up to 345.5 milliseconds, with a pause occurring on average about three times per second. These pauses would be noticed by a human user and are unsuitable for real-time applications, which precludes CHERIvoke from practical deployment. Additionally, as evaluated further below, pause times for the CHERIvoke algorithm scale linearly with the application's memory footprint. The `xalancbmk` result is with a maximum memory footprint of approximately 800 MiB; real-world systems that provide heap temporal safety can be expected to handle memory footprints dozens of times that size [112], which is clearly not possible with the CHERIvoke algorithm.

The store-side algorithm only pauses the application to sweep pages that have experienced capability stores concurrently with the initial phase. It reduces pause times significantly relative to CHERIvoke, but, as discussed further below, its pause times still scale with the application's memory footprint. The load-side algorithm has orders-of-magnitude lower pause times and performs a constant amount of work with the application paused, regardless of the application's memory footprint.

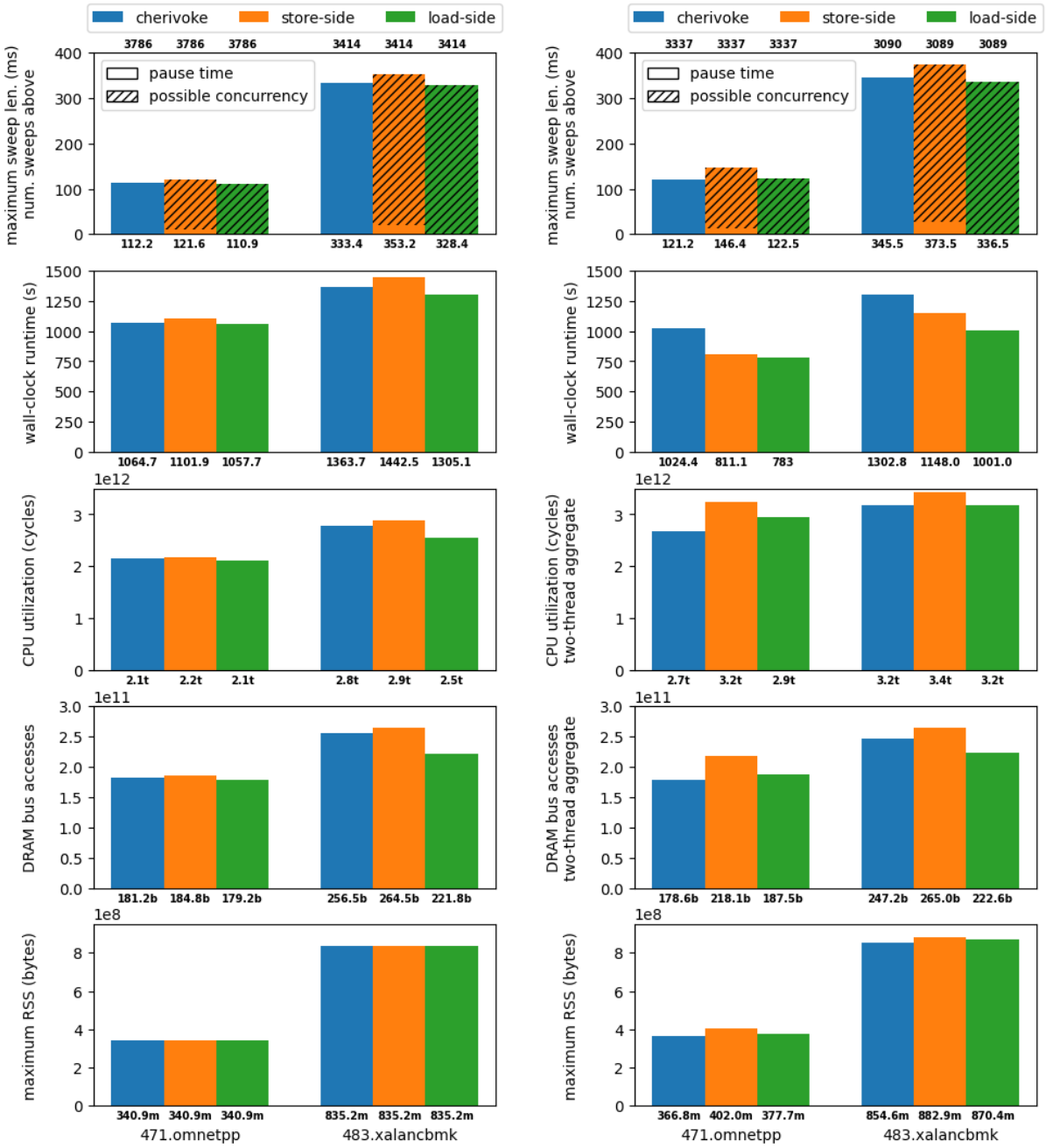
These results demonstrate that store-side and load-side revocation offer clear benefits relative to CHERIvoke in terms of pause times. Load-side revocation in particular is potentially suitable for real-time or interactive applications and enterprise applications with large memory footprints. The most significant difference in worst-case pause times is a reduction of over 11,000x between the CHERIvoke and load-side case for `xalancbmk`. In general, the relative difference in pause times between the revocation methods depends on both the number of capability-containing pages owned by the application and to what extent the application writes capabilities to those pages concurrently with the revocation pass. I perform further evaluation of the scaling behavior of pause times below.

The sweep counts above the top axis reveal that only `xalancbmk` and `omnetpp` use the heap intensely enough to cause a large number of revocation passes. `libquantum` and `xalancbmk` perform one more revocation pass in the CHERIvoke case relative to the concurrent algorithms. In those cases, the applications finish executing while a concurrent phase of a revocation pass is in progress.

The middle axis measures the applications' wall-clock runtimes. These are essentially the same across all benchmarks except `omnetpp` and `xalancbmk`, because the others do not perform enough revocation passes for the differences to be detected. However, for `omnetpp` and `xalancbmk` the differences are significant. For example, the runtime of `omnetpp` with load-side revocation is 23.5% less than with the CHERIvoke algorithm. In these cases, the difference in wall-clock runtimes is due to the significant discrepancy in pause times as well as other characteristics of the sweeping algorithms, which I will discuss further below.

The bottom axis measures maximum resident set size (RSS), or memory footprint. Because of the trade-off between memory footprint and other performance costs for sweeping revocation, maximum RSS values should be as similar as possible between the revocation algorithms to ensure fair comparison. The axis shows that the values are indeed similar. Some benchmarks have a slightly increased maximum RSS for the concurrent algorithms, with the store-side maximum RSS higher than the load-side case. This is because the more time an application runs concurrently with the revoker, the higher its

memory footprint will be.



(a) Results without an offload thread.

(b) Results with an offload thread.

Figure 5.6: Detailed performance measurements with and without an offload thread for the worst-case `omnetpp` and `xalancbmk` benchmarks. Without an offload thread, the load-side barrier algorithm has lower CPU utilization and fewer DRAM accesses than the CHERIvoke baseline. With an offload thread, relative resource utilization depends on application characteristics.

Offload threads Figures 5.6a and 5.6b present additional measurements for the worst-case `omnetpp` and `xalancbmk` benchmarks, both without and with offload threads respec-

tively. They are similar in layout to Figure 5.5. The top axis has been augmented to display the length of the entire longest revocation pass, with the portion that can happen concurrently displayed using a hatched pattern. The new third and fourth axes from the top show CPU utilization and DRAM bus accesses, obtained by measuring Arm Performance Monitoring Unit counters with the `pmcstat` tool. CPU utilization represents the number of unhalted clock cycles spent running the application, while DRAM bus accesses represent the number of cache-hierarchy misses. In Figure 5.6b, these axes aggregate counter values from both the application and offload threads.

I first consider the results of Figure 5.6a. Unlike the data presented previously in Figure 5.5, these data come from runs without an offload thread enabled in the Cornucopia wrapper. As a result, all three sweeping revocation algorithms effectively perform their entire revocation passes with the application paused, because the SPEC benchmarks are single-threaded and the `cheri_revoke()` system call consumes the calling thread. This allows for fair comparison of properties of the algorithms that are not related to parallel execution.

In the non-offload case, the number of revocation passes and maximum RSS values are identical across the revocation algorithms for each benchmark, because the application is not allowed to execute in parallel with the revoker. Also, unlike in the offload case, the Cornucopia wrapper module calls `malloc_wrapper_underlying_allocation()` to validate capabilities passed to `free()` before adding them to the quarantine buffer. In the offload case, the wrapper adds capabilities passed to `free()` directly to the quarantine buffer, and they are validated by the offload thread before it performs a revocation pass. Because `malloc_wrapper_underlying_allocation()` returns a capability to an underlying allocation, which may have a length greater than the passed-in capability, quarantine buffers fill more quickly in the non-offload case. This explains the facts that maximum RSS values are lower, the numbers of revocation passes are higher, and the lengths of revocation passes are shorter in the non-offload case relative to the offload case.

Figure 5.6a reveals that, without any parallel execution, the load-side algorithm performs better than CHERIvoke and the store-side algorithm performs worse than CHERIvoke in all observed metrics: worst-case sweep length and pause time, wall-clock runtime, CPU utilization, and DRAM bus accesses. The worse performance of the store-side algorithm can be explained by the fact that, even when no pages have experienced capability stores concurrently with the first phase of a revocation pass, the algorithm must still iterate over the process' entire virtual address space a second time to verify that no concurrent capability stores took place. On the other hand, the better performance of the load-side algorithm can be explained by the fact that its technique for iterating over a process' virtual address space requires less state propagation and fewer lock acquisitions than the store-side and CHERIvoke algorithms.

Another interesting feature of the chart is that the results of dividing the reported CPU utilizations by the 2.5 GHz processor clock speed yields values that are less than the observed wall-clock times. This effect might be explained by time spent with the processor core halted while waiting to acquire locks for kernel data structures, or by kernel threads unrelated to the application being scheduled on the core.

Figure 5.6b provides some additional data for `omnetpp` and `xalancbmk` in the offload case. Relative to the non-offload case, parallelism in the offload case lowers wall-clock runtimes at the expense of increased CPU utilization and DRAM bus accesses. The CHERIvoke algorithm experiences lower wall-clock runtimes in the offload case, even

though the algorithm doesn't allow parallel execution, because the wrapper's validation of capabilities passed to `free()` is performed in the offload thread.

The store-side algorithm has worse performance than CHERIvoke in terms of CPU utilization and DRAM bus accesses, which is expected given its design. Interestingly, the performance of the load-side algorithm relative to CHERIvoke depends on characteristics of the application. While the CHERIvoke algorithm must experience some overhead associated with spawning and synchronizing with a second thread in the offload case, it should experience the minimum possible cache contention between the application and the offload threads, because it does not allow the application to run in parallel with the revoker. In the `xalancbmk` case, however, the load-side algorithm results in fewer DRAM bus accesses than the CHERIvoke algorithm. This might be explained by the load-side algorithm's improved cache locality properties. Load-side barriers in the fault-handling context can prefetch data used by the application into caches for the core running the application thread. With CHERIvoke and the store-side algorithm, on the other hand, the revoker runs solely on a separate core and loads data into caches independently of application behavior. For `xalancbmk`, the load-side algorithm also has essentially the same CPU utilization as CHERIvoke. On the other hand, in the `omnetpp` case the load-side algorithm results in approximately 5% more DRAM bus accesses and approximately 7% higher CPU utilization compared to CHERIvoke. These data suggest that the CPU utilization and DRAM bus accesses of CHERIvoke and the load-side barrier algorithm are similar, and which performs better is determined by application behavior.

Pause times In the above descriptions of the sweeping revocation algorithms and performance evaluation, I asserted that the worst-case pause times for the CHERIvoke and store-side algorithms scale linearly with the memory footprint of the running application, while the worst-case pause times for the load-side algorithm stay constant. Figure 5.7 demonstrates that this is indeed the case. It measures the worst-case pause times across the algorithms for the *test* input configuration of `xalancbmk` with the wrapper configured to use no offload thread and quarantine buffer thresholds of 1MiB, 5MiB, 10MiB, 15MiB, and 20MiB. The application's maximum RSS increases identically across the sweeping revocation algorithms as the wrapper's quarantine buffer threshold increases because there is no offload thread. As expected, maximum pause times for the CHERIvoke algorithm and store-side algorithm scale with increasing maximum RSS sizes, albeit at different rates, while maximum pause times for the load-side algorithm remain constant.

5.4.3 Cornucopia wrapper module

In this subsection, I evaluate the source-code compatibility benefits and performance costs of the generic Cornucopia wrapper module relative to direct integration. The `snmalloc` and `dlmalloc` allocators were adapted to integrate with Cornucopia directly by other members of my research group; I adapted these allocators to use my generic wrapper module, and I also made bug-fixes and refinements to the directly integrated versions to facilitate comparability. CheriBSD's default memory allocator, `jemalloc`, has a complex and highly-optimized codebase. Adding direct Cornucopia integration to it was determined not to be worth the effort by my colleagues. However, I was able to add support for the wrapper module trivially. This anecdotal evidence hints at the utility of the wrapper module's minimal source-code compatibility requirements. Below, I more rigorously

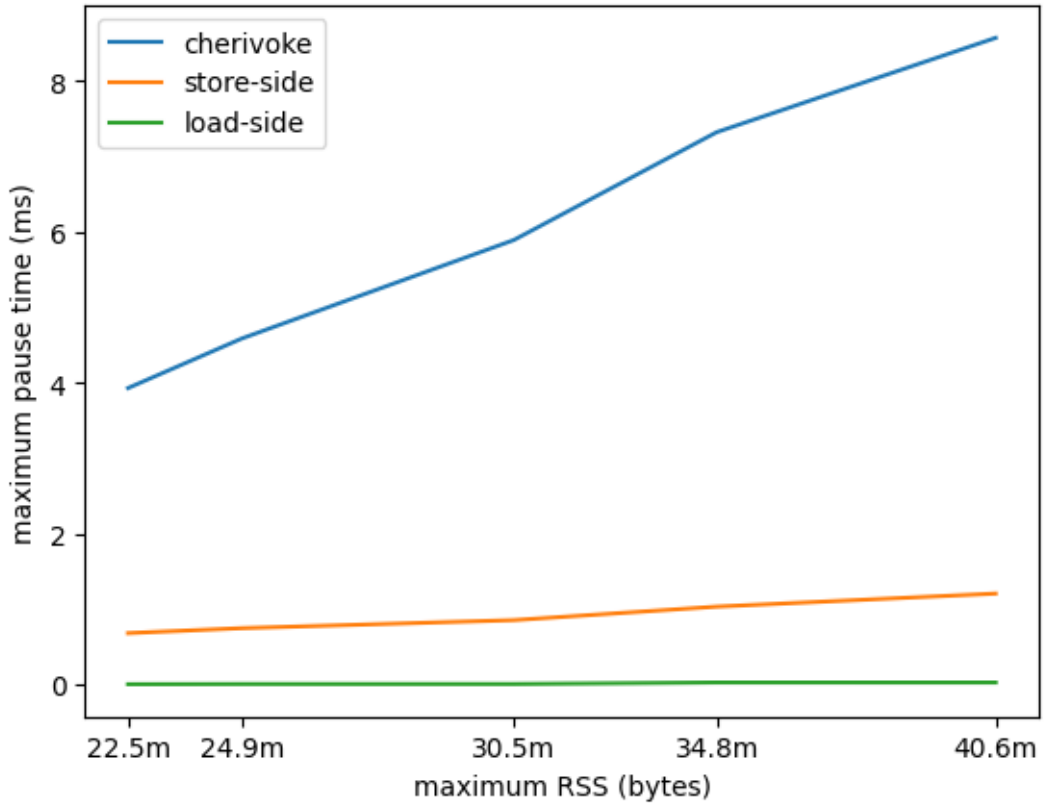


Figure 5.7: Varying Cornucopia wrapper quarantine buffer sizes, which affects application RSS, and observing changes in worst-case pause times for different revocation algorithms in runs of the *test* configuration of `xalancbmk`. The CHERIvoke and store-side barrier algorithm pause times scale linearly with application RSS, while the load-side barrier algorithm pause times stay constant.

quantify the differences in source-code changes required and runtime performance between direct Cornucopia integration and use of the wrapper module for `snmalloc` and `dlmalloc`.

5.4.3.1 Source-code compatibility

As in Chapter 4, I use the `cloc` tool to measure source-code changes between versions of the two allocators. The baseline for each is an allocator that meets the prerequisites for temporal safety outlined earlier in this chapter. The experimental cases are adaptations to support direct integration with Cornucopia and integration via the generic wrapper module.

Table 5.1 presents data on files and lines of C/C++ code added, removed, or modified between the baseline and experimental cases. The *direct* configuration suffix indicates direct integration with Cornucopia, while the *wrapper* suffix indicates use of the wrapper module. The data are presented as absolute numbers and as a percentage of the total number of files and lines in the baseline case. They clearly demonstrate that the source-code compatibility burden of using the wrapper module is significantly less than integrating with Cornucopia directly. For `snmalloc`, the number of lines of code that must be modified to use the wrapper module is 0.94% of that required for direct integration; for `dlmalloc`, it is 2.21%. This large discrepancy is because direct integration with Cornucopia requires deep

Name	Files modified (%)	Lines modified (%)
snmalloc-direct	15 (24.59)	1272 (18.8)
snmalloc-wrapper	2 (3.28)	12 (0.18)
dlmalloc-direct	3 (100)	904 (31.51)
dlmalloc-wrapper	2 (66.67)	20 (0.70)

Table 5.1: Source-code compatibility metrics for `snmalloc` and `dlmalloc` using direct Cornucopia integration and the generic wrapper module. Using the wrapper requires significantly less source-code modification than direct integration.

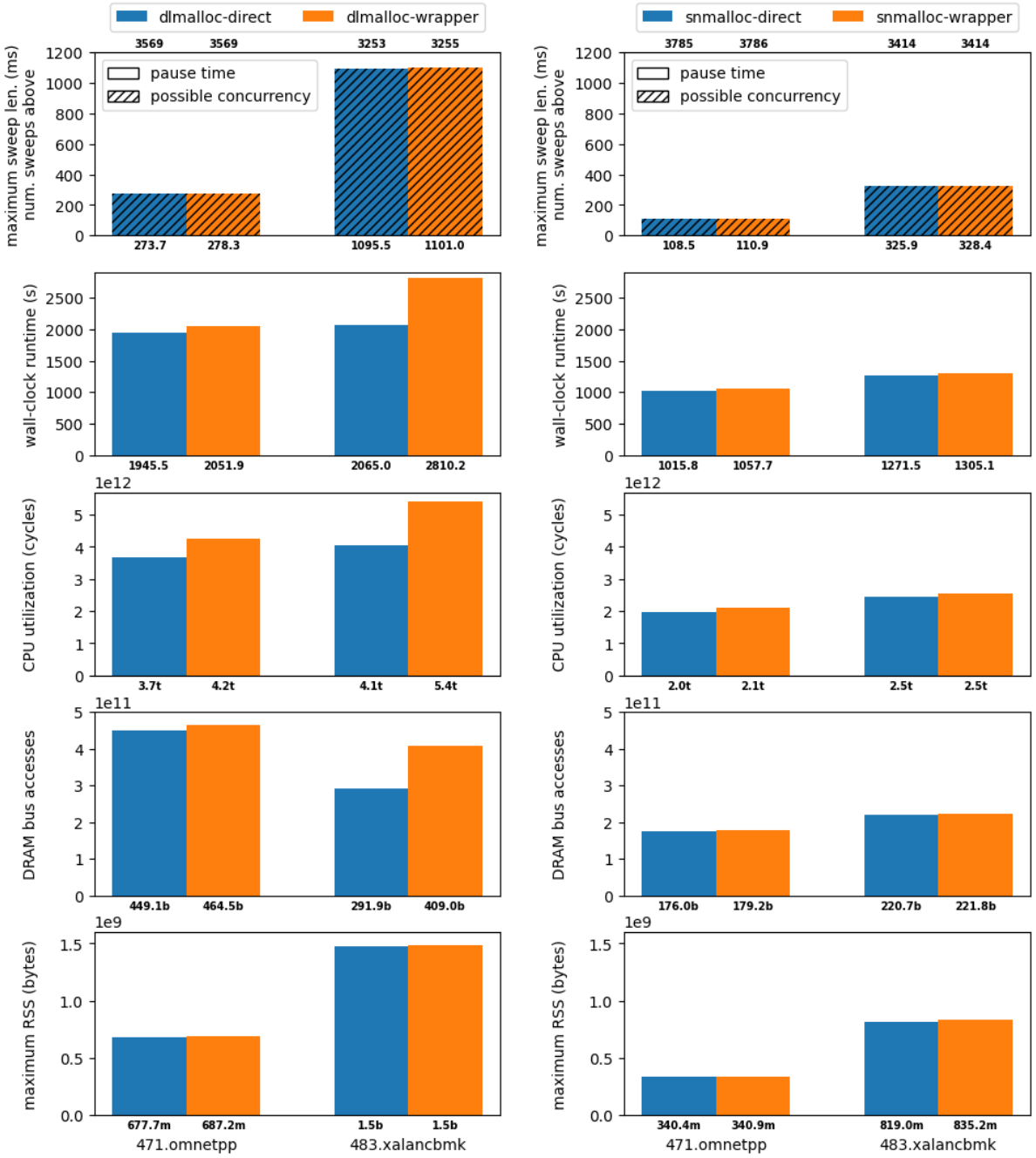
modification of allocation routines and allocator-internal data structures, while use of the generic wrapper only requires the function `malloc_wrapper_underlying_allocation()` to be declared and defined. `malloc_wrapper_underlying_allocation()` can often be implemented by calling or refactoring existing functions, because similar logic already exists in the allocator to validate inputs to `free()` and implement `malloc_usable_size()`. Interestingly, the absolute number of lines changed required for both the direct and wrapper integrations are roughly similar for `snmalloc` and `dlmalloc`. The percentage values are relatively lower for `snmalloc` because it is a more sophisticated and optimized memory allocator with a larger codebase.

5.4.3.2 Performance

To measure the performance costs of the generic wrapper module, I ran the *ref* workloads of the `omnetpp` and `xalancbmk` benchmarks with heap temporal safety using the directly integrated and wrapper-supporting adaptations of both `snmalloc` and `dlmalloc`. The direct integrations for both allocators did not support an offload thread, so the wrapper's offload thread was disabled. The load-side sweeping revocation algorithm was used for all configurations under test, and all configurations used a fixed quarantine buffer size threshold of 20 MiB.

For comparability, I modified the direct integrations to match the generic wrapper as closely as possible in terms of incrementing the quarantine buffer size on calls to `free()`. Some minor bookkeeping discrepancies remain, but I have ensured that they result in the wrapper incrementing the quarantine buffer size by more than the directly integrated adaptation. As a result, the wrapper configuration may perform more revocation passes than the direct integration; these results can thus be considered an upper bound on the wrapper's performance cost.

Figure 5.8a presents results for the `dlmalloc` runs using the same chart axes as Figures 5.6a and 5.6b. The direct-integration configuration is in blue and the generic wrapper configuration is in orange. The maximum RSS and sweep length are nearly identical for the wrapper configuration and direct integration, suggesting that the wrapper's extra memory requirement is not onerous. However, the wrapper's effects on wall-clock runtime, CPU utilization, and DRAM bus accesses are significant. The wall-clock runtime for `xalancbmk` is 36.09% higher with the wrapper than with direct integration, the CPU utilization is 31.71% higher, and the number of DRAM bus accesses is 40.12% higher. These results suggest that the operations to validate and re-derive capabilities in `dlmalloc`'s implementation of `malloc_wrapper_underlying_allocation()` are relatively costly.



(a) Results for `dlmalloc`.

(b) Results for `snmalloc`.

Figure 5.8: Performance results for direct and wrapper-based Cornucopia integration on the worst-case `omnetpp` and `xalancbmk` benchmarks. Overhead for the wrapper depends on the underlying allocator's design and application characteristics. Contemporary memory allocators like `snmalloc` can use the wrapper as a replacement for direct integration with very little overhead.

Unlike `xalancbmk`, `omnetpp` does not experience as much of a difference between the wrapper configuration and direct integration. This can be explained by the fact that application behavior can have a significant effect on the relative cost of using the Cornucopia wrapper and on sweeping capability revocation in general. For example, an

application that frees a large number of small allocations will perform more bookkeeping for the quarantine buffer, and, if the Cornucopia wrapper is in use, make more calls to `malloc_wrapper_underlying_allocation()` than an application that frees a small number of large allocations.

Figure 5.8b presents similar results for `snmalloc`. For all measured metrics across both benchmarks, the wrapper configuration achieves results within approximately 5% of the direct integration configuration. The higher wall-clock runtime and CPU utilization can be explained by the additional work that the wrapper must perform to validate capabilities passed to `free()`. When working with `snmalloc` as the underlying allocator, the wrapper does not result in a significant increase in DRAM traffic.

Interestingly, the `dlmalloc` results feature significantly different behavior for `xalancbmk` and `omnetpp`, while the `snmalloc` results do not. This is because `snmalloc` is a contemporary and highly optimized slab allocator, and its operations to validate and re-derive capabilities in `malloc_wrapper_underlying_allocation()` are much less costly than those of `dlmalloc`, which is an older boundary-tag allocator. These results suggest that modern allocators such as `snmalloc` can use the wrapper as a replacement for direct integration with very little performance overhead. For legacy allocators, depending on application behavior, the wrapper may or may not result in a significant overhead. In performance-critical settings where the wrapper results in significant overhead, the wrapper may still be useful as an initial low-effort step towards direct Cornucopia integration.

5.5 Discussion

The above analysis and evaluation suggest that the hypotheses set out at the beginning of the chapter can be answered affirmatively. Efficacy results from running the Juliet test suite on the Morello SoC with Cornucopia demonstrate that it is indeed possible to implement sweeping capability revocation on real CHERI hardware that mitigates heap temporal safety vulnerabilities in practice. The store-side sweeping revocation performs more work relative to the CHERIvoke baseline, but results in shorter pause times and wall-clock runtimes in multi-threaded scenarios. The load-side sweeping revocation algorithm, which makes use of new architectural features on Morello, facilitates extreme reductions in application pause times and wall-clock runtimes. Its pause times are also independent of application memory footprint. These features make the load-side algorithm, unlike the CHERIvoke baseline or the store-side algorithm, potentially suitable for real-time, interactive, and enterprise-scale applications. The generic Cornucopia wrapper module requires significantly less source-code modification than direct integration. Its performance overhead is low for modern allocators, though certain application behavior can result in significant overheads for legacy allocators.

5.5.1 Future work

There is a number of promising directions for future work involving sweeping capability revocation on Morello. These include:

Further analysis of application and allocator behavior The above analysis and evaluation identified various factors that can influence the performance of sweeping capability revocation. These include an application's memory footprint and distribution of capabilities, the rate at which an application frees memory and the size

of those freed regions, and a memory allocator's design. While these factors were discussed and measured sufficiently to address this chapter's hypotheses, future work should perform further analyses to yield more insight about the performance cost of sweeping revocation. This may include collecting metrics on heap fragmentation and allocation patterns, the number of pages and capabilities examined by the revoker, and the number of store- or load-side barriers triggered. Similarly, more detailed information about the distribution of revocation pass sweep lengths and pause times might usefully be collected.

Performance relative to CheriABI baseline Because of current limitations of the Morello SoC platform, experiments in this chapter were designed to compare sweeping revocation algorithms rather than to measure the overall cost of enabling sweeping revocation. Once the limitations have been addressed, future work should measure this overall cost and iterate on the system's design to reduce it as much as possible. A number of optimization opportunities could be explored, such as when to perform global TLB invalidations, how to reduce kernel data structure locking, parallelizing the revoker itself, and, for the load-side algorithm, using heuristics to scan multiple pages in load-side barriers and reduce the overall number of faults taken. Ultimately, the cost of sweeping revocation must be made low enough to reasonably deploy temporal safety with Cornucopia as an always-on mitigation.

Further optimization opportunities There are various further optimization opportunities for sweeping capability revocation that should be explored in depth in future work. These include:

- The revoker's page-processing loop could be optimized with hand-written assembly, perhaps using conditional execution to reduce branch mispredictions and prefetching to improve cache performance.
- While the `CLoadTags` instruction suggested in the CHERIvoke paper is non-temporal, Morello's concrete `ldct` implementation loads capabilities into the cache hierarchy before returning their tags. Cache pollution may be reduced by changing the revoker to route `ldct` instructions through a memory map that is marked non-cacheable, so that the instructions are effectively made non-temporal.
- It may be possible to improve the performance of revocation passes by using broadcast `tlbi` instructions rather than explicit inter-processor interrupts to perform global TLB invalidations.
- The Morello SoC might be configured to store capability tags in a separate tag memory rather than DRAM error-correcting-code bits to determine whether this makes a difference for performance.
- The performance of sweeping revocation might be improved by composing it with a form of Arm's Memory Tagging Extension (See Appendix D.6 of the CHERI ISA manual [118]).

Multi-threaded applications Cornucopia currently supports a form of epoch-based reclamation [44] that allows multiple threads of an application's memory allocator to detect and benefit from revocation sweeps performed by other threads. Future

work might investigate the use of this feature in depth, which could facilitate the use of Cornucopia for contemporary enterprise-scale applications.

Chapter 6

Conclusions

This dissertation made a number of contributions to the state of the art in memory-safety protections with CHERI for low-level languages like C and C++. It introduced novel ideas and supported them with systematic analysis, complex prototypes based on real-world software, and sophisticated evaluation on Arm's Morello platform.

6.1 Contributions

In Chapter 3, I developed the memory-operations framework (MOF) for reasoning about memory-safety mitigations and the types of attacks they protect against. I applied this framework to analyze the memory-safety security properties of CheriABI and to examine how it would protect against a set of real-world exploit chains. The exploit chains relied heavily on weaknesses in contemporary language interpreters, which had not previously been adapted to support pure-capability compilation, and on temporal-safety vulnerabilities, for which a practical mitigation using CHERI capabilities had not been developed.

In Chapter 4, I described adapting the JavaScriptCore interpreter, including its baseline just-in-time compiler, to support pure-capability compilation on Morello. While language interpreters with JIT compilers have significantly different characteristics than typical applications, I determined that they can be adapted to support pure-capability compilation without affecting their functionality. I characterized the interesting and generalizable aspects of the significant engineering effort required to adapt JSC, and I quantified the source-code compatibility burden as comparable to that of adapting similar types of software. I also determined that, while pure-capability compilation does not automatically grant language interpreters the same memory-safety security properties that it does for more typical applications, straightforward changes to interpreter memory allocators, JIT compilers, and build systems can bring their security properties up to par with CheriABI.

In Chapter 5, I described the design, implementation, and evaluation of a fully elaborated system for heap temporal safety with sweeping CHERI capability revocation. I introduced the novel store-side and load-side barrier algorithms for sweeping capability revocation, which significantly improve performance relative to the CHERIvoke baseline. I also introduced the Cornucopia subsystem of the CheriBSD kernel and userspace heap memory allocators that were adapted to integrate with it directly or to use a generic wrapper module. I implemented and evaluated these components on Morello, collecting the first performance results from the system-on-chip to be published. I determined that,

by leveraging architectural support for capability load generations, the load-side barrier algorithm delivers extreme performance improvements relative to the CHERIvoke baseline. The load-side algorithm's performance characteristics, particularly low pause times that remain small and constant even with increasing application memory footprints, make it potentially suitable for use in real-time, interactive, and enterprise-scale applications. For contemporary memory allocators, the generic Cornucopia wrapper module provides significant source-code compatibility benefits at a very low performance cost.

6.2 Future work

In addition to work that builds directly on the results of this dissertation, there are various exciting future directions for the CHERI project as a whole. Ongoing work to adapt the CheriBSD kernel to support pure-capability compilation lays the foundation for general-purpose systems in which all memory accesses are made via CHERI capabilities and a small TCB delivers memory safety for all software. Furthermore, solutions to effectively integrate fine-grained compartmentalization supported by CHERI architectural features with existing codebases can mitigate or limit the scope of a wide range of vulnerabilities, even those that involve logic errors. Recent efforts to adapt a wide range of desktop software to support pure-capability compilation, combined with the availability of Morello boards, mean that using CHERI machines as general-purpose workstations is nearly a reality. And mature performance results from the Morello SoCs will ultimately determine whether superscalar capability machines are suitable for widespread adoption.

Bibliography

[1] Martin Abadi, Mihai Budiu, and Jay Ligatti. "Control-Flow Integrity: Principles, Implementations, and Applications". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS 2005. Alexandria, VA, USA, p. 14 (cit. on p. 18).

[2] Sam Ainsworth and Timothy M. Jones. "MarkUs: Drop-in use-after-free prevention for low-level languages". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2020, pp. 578–591. DOI: 10.1109/SP40000.2020.00058 (cit. on pp. 20, 70).

[3] Periklis Akritidis. "Cling: A Memory Allocator to Mitigate Dangling Pointers". In: USENIX Security 2010, p. 16 (cit. on p. 20).

[4] AlDanial. *cloc*. Feb. 24, 2022. URL: https://github.com/AlDanial/cloc (visited on 02/24/2022) (cit. on p. 59).

[5] James P. Anderson. *Computer Security Technology Planning Study*. USAF Electronic Systems Division, Oct. 1972 (cit. on p. 36).

[6] John Aycock. "A brief history of just-in-time". In: *ACM Computing Surveys* 35.2 (June 2003), pp. 97–113. DOI: 10.1145/857076.857077 (cit. on p. 28).

[7] Henry G. Baker. "List processing in real time on a serial computer". In: *Communications of the ACM* 21.4 (Apr. 1978), pp. 280–294. DOI: 10.1145/359460.359470 (cit. on p. 34).

[8] Saam Barati and Yusuke Suzuki. *JSC - ES6*. WebKit. June 5, 2017. URL: https://webkit.org/blog/7536/jsc-loves-es6/ (visited on 05/22/2022) (cit. on p. 58).

[9] Ian Beer. *Project Zero: A very deep dive into iOS Exploit chains found in the wild*. Project Zero. Aug. 29, 2019. URL: https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html (visited on 01/13/2020) (cit. on pp. 42, 43).

[10] Ian Beer. *Project Zero: Splitting atoms in XNU*. Project Zero. Apr. 1, 2019. URL: https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html (visited on 04/06/2019) (cit. on pp. 21, 66).

[11] Emery D Berger and Benjamin G Zorn. "DieHard: Probabilistic Memory Safety for Unsafe Languages". In: PLDI 2006, p. 11 (cit. on p. 20).

[12] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. ASIACCS. Hong Kong, China: ACM Press, 2011, p. 30. DOI: 10.1145/1966913.1966919 (cit. on p. 18).

[13] Hans-Juergen Boehm, Alan J Demers, and Scott Shenker. "Mostly Parallel Garbage Collection". In: *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. (), p. 8 (cit. on pp. 34, 73, 78).

[14] Hans-Juergen Boehm and Mark Weiser. "Garbage collection in an uncooperative environment". In: *Software: Practice and Experience* 18.9 (Sept. 1988), pp. 807–820. DOI: 10.1002/spe.4380180902 (cit. on pp. 20, 32, 33).

[15] Tim Boland and Paul E. Black. "Juliet 1.1 C/C++ and Java Test Suite". In: *IEEE Computer* 45.10 (Oct. 2012), pp. 88–90. DOI: 10.1109/MC.2012.345 (cit. on p. 93).

[16] Emma Bowman. "After Data Breach Exposes 530 Million, Facebook Says It Will Not Notify Users". In: *NPR* (Apr. 9, 2021). URL: https://www.npr.org/2021/04/09/986005820/after-data-breach-exposes-530-million-facebook-says-it-will-not-notify-users (cit. on p. 9).

[17] R. J. Brache. *Introduction to Intel Memory Protection Extensions*. URL: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions (visited on 05/03/2019) (cit. on p. 21).

[18] Nicholas P Carter, Stephen W Keckler, and William J Dally. "Hardware Support for Fast Capability-based Addressing". In: ASPLOS 1994, p. 9 (cit. on p. 19).

[19] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-Flow Integrity". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI. USA: USENIX Association, 2006, pp. 147–160. ISBN: 1-931971-47-1 (cit. on p. 19).

[20] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and Communications Security - CCS '10*. CCS. Chicago, Illinois, USA: ACM Press, 2010, p. 559. DOI: 10.1145/1866307.1866370 (cit. on p. 18).

[21] Long Cheng, Hans Liljestrand, Thomas Nyman, Yu Tsung Lee, Danfeng Yao, Trent Jaeger, and N. Asokan. *Exploitation Techniques and Defenses for Data-Oriented Attacks*. Mar. 24, 2019. URL: http://arxiv.org/abs/1902.08359 (visited on 01/05/2022) (cit. on p. 19).

[22] Catalin Cimpanu. *Microsoft: 70 percent of all security bugs are memory safety issues*. ZDNet. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (visited on 04/28/2019) (cit. on pp. 9, 16).

[23] *Control Flow Integrity*. Android Open Source Project. URL: https://source.android.com/devices/tech/debug/cfi (visited on 12/31/2021) (cit. on pp. 9, 16).

[24] *Control Flow Integrity — Clang 13 documentation*. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html (visited on 01/05/2022) (cit. on p. 18).

[25] *CTSRD-CHERI/CheriBSD*. Dec. 13, 2021. URL: https://github.com/CTSRD-CHERI/cheribsd (visited on 01/02/2022) (cit. on pp. 10, 23).

[26] *CTSRD-CHERI/FreeRTOS*. Jan. 2, 2021. URL: https://github.com/CTSRD-CHERI/FreeRTOS (visited on 01/02/2022) (cit. on p. 10).

[27] *CTSRD-CHERI/llvm-project*. Nov. 23, 2021. URL: https://github.com/CTSRD-CHERI/llvm-project (visited on 01/02/2022) (cit. on p. 10).

[28] *CTSRD-CHERI/postgres*. Dec. 4, 2020. URL: https://github.com/CTSRD-CHERI/postgres (visited on 01/02/2022) (cit. on p. 10).

[29] *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/ (visited on 05/08/2019) (cit. on p. 38).

[30] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. "Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers". In: USENIX Security 2017, p. 18 (cit. on pp. 20, 70).

[31] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. "The Performance Cost of Shadow Stacks and Stack Canaries". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. AsiaCCS 15. Singapore Republic of Singapore: ACM, Apr. 14, 2015, pp. 555–566. DOI: 10.1145/2714576.2714635 (cit. on p. 18).

[32] Brooks Davis, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Robert N. M. Watson, Stacey Son, Jonathan Woodruff, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, and Nathaniel Wesley Filardo. "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*. ASPLOS 2019. Providence, RI, USA: ACM Press, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042 (cit. on pp. 10, 23, 24, 38, 51, 88).

[33] Alan Demmers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. "Combining generational and conservative garbage collection: framework and implementations". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '90*. San Francisco, California, United States: ACM Press, 1990, pp. 261–269. DOI: 10.1145/96709.96735 (cit. on pp. 29, 34).

[34] Joe Devietti, Colin Blundell, Milo M K Martin, and Steve Zdancewic. "HardBound: Architectural Support for Spatial Safety of the C Programming Language". In: ASPLOS 2008 (cit. on p. 19).

[35] Dinakar Dhurjati and Vikram Adve. "Backwards-compatible array bounds checking for C with very low overhead". In: *Proceeding of the 28th international conference on Software engineering - ICSE '06*. ICSE 2006. Shanghai, China: ACM Press, 2006, p. 162. DOI: 10.1145/1134285.1134309 (cit. on p. 19).

[36] Dinakar Dhurjati and Vikram Adve. "Efficiently Detecting All Dangling Pointer Uses in Production Servers". In: *International Conference on Dependable Systems and Networks (DSN'06)*. DSN 2006. Philadelphia, PA, USA: IEEE, 2006, pp. 269–280. DOI: 10.1109/DSN.2006.31 (cit. on p. 20).

[37] Damien Doligez and Georges Gonthier. "Portable, unobtrusive garbage collection for multiprocessor systems". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*. Portland, Oregon, United States: ACM Press, 1994, pp. 70–83. DOI: 10.1145/174675.174673 (cit. on p. 34).

[38] Damien Doligez and Xavier Leroy. "A concurrent, generational garbage collector for a multithreaded implementation of ML". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*. Charleston, South Carolina, United States: ACM Press, 1993, pp. 113–123. DOI: 10.1145/158511.158611 (cit. on p. 34).

[39] Thomas F. Dullien. "Weird machines, exploitability, and provable unexploitability". In: *IEEE Transactions on Emerging Topics in Computing* (2018). DOI: 10.1109/TETC.2017.2785299 (cit. on p. 38).

[40] *ECMAScript 2015 Language Specification – ECMA-262 6th Edition*. URL: https://262.ecma-international.org/6.0/ (visited on 05/22/2022) (cit. on pp. 29, 58).

[41] *Electric Fence*. URL: https://elinux.org/Electric_Fence (cit. on p. 20).

[42] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". BSDCan 2006 (cit. on pp. 89, 93).

[43] *Firing up the Ignition interpreter · V8*. URL: https://v8.dev/blog/ignition-interpreter (visited on 05/20/2022) (cit. on p. 57).

[44] Keir Fraser. *Practical lock-freedom*. 579. University of Cambridge Computer Laboratory, Feb. 2004, p. 116. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf (cit. on p. 105).

[45] *Gigacage — phakeobj*. URL: https://phakeobj.netlify.app/posts/gigacage/ (visited on 02/22/2022) (cit. on p. 65).

[46] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *Proceedings 2017 Network and Distributed System Security Symposium*. NDSS. San Diego, CA: Internet Society, 2017. DOI: 10.14722/ndss.2017.23271 (cit. on p. 18).

[47] Matthew Gretton-Dann. *Arm A-Profile Architecture Developments 2018: Armv8.5-A*. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a (visited on 05/10/2019) (cit. on p. 21).

[48] Samuel Groß. "Attacking Client-Side JIT Compilers (v2)". URL: https://saelo.github.io/presentations/blackhat_us_18_attacking_client_side_jit_compilers.pdf (cit. on p. 43).

[49] Samuel Groß. *Project Zero: JITSploitation I: A JIT Bug*. Project Zero. Sept. 1, 2020. URL: https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html (visited on 03/02/2022) (cit. on pp. 21, 43, 65, 66).

[50] Niranjan Hasabnis, Ashish Misra, and R. Sekar. "Light-weight bounds checking". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization - CHO '12*. San Jose, California: ACM Press, 2012, p. 135. DOI: 10.1145/2259016.2259034 (cit. on p. 19).

[51] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *2016 IEEE Symposium on Security and Privacy (SP)*. SSP. San Jose, CA: IEEE, May 2016, pp. 969–986. DOI: 10.1109/SP.2016.62 (cit. on p. 19).

[52] *Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector*. WebKit. Jan. 20, 2017. URL: https://webkit.org/blog/7122/introducing-riptide-webkits-retreating-wavefront-concurrent-garbage-collector/ (visited on 02/21/2022) (cit. on p. 29).

[53] Dejice Jacob and Jeremy Singer. "Capability Boehm: challenges and opportunities for garbage collection with capability hardware". In: *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, Feb. 25, 2022, pp. 81–87. DOI: 10.1145/3516807.3516823 (cit. on p. 58).

[54] *JavaScriptCore – WebKit*. URL: https://trac.webkit.org/wiki/JavaScriptCore (visited on 01/20/2022) (cit. on p. 28).

[55] *jefree doesn't check that the capability is valid before freeing it · Issue #393 · CTSRD-CHERI/cheribsd*. GitHub. URL: https://github.com/CTSRD-CHERI/cheribsd/issues/393 (visited on 04/01/2022) (cit. on p. 88).

[56] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. "Efficient Tagged Memory". In: *2017 IEEE International Conference on Computer Design (ICCD)*. ICCD 2017. Boston, MA: IEEE, Nov. 2017, pp. 641–648. DOI: 10.1109/ICCD.2017.112 (cit. on p. 26).

[57] Nicolas Joly, Saif ElSherei, and Saar Amar. *Security Analysis of CHERI ISA*. Microsoft, Feb. 7, 2022. URL: https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/ (cit. on p. 38).

[58] Richard W M Jones and Paul H J Kelly. "Backwards-compatible bounds checking for arrays and pointers in C programs". In: AADEBUG 1997. Linkoping, Sweden, p. 14 (cit. on p. 19).

[59] *Juliet Test Suite Documents*. Software Assurance Reference Dataset. URL: https://samate.nist.gov/SRD/around.php#juliet_documents (visited on 05/09/2019) (cit. on p. 93).

[60] Vinay Katoch. *Bypassing ASLR/DEP*. URL: https://www.exploit-db.com/docs/english/17914-bypassing-aslrdep.pdf (cit. on p. 18).

[61] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. "DangSan: Scalable Use-after-free Detection". In: *Proceedings of the Twelfth European Conference on Computer Systems - EuroSys '17*. EuroSys 2017. Belgrade, Serbia: ACM Press, 2017, pp. 405–419. DOI: 10.1145/3064176.3064211 (cit. on pp. 20, 71).

[62] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. CCS 2013. Berlin, Germany: ACM Press, 2013, pp. 721–732. DOI: 10.1145/2508859.2516713 (cit. on p. 19).

[63] Doug Lea. *A Memory Allocator*. URL: http://gee.cs.oswego.edu/dl/html/malloc.html (visited on 04/11/2022) (cit. on pp. 89, 93).

[64] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. "Preventing Use-after-free with Dangling Pointers Nullification". In: *Proceedings 2015 Network and Distributed System Security Symposium*. NDSS 2015. San Diego, CA: Internet Society, 2015. DOI: 10.14722/ndss.2015.23238 (cit. on pp. 20, 70, 71).

[65] Henry Lieberman and Carl Hewitt. "A real-time garbage collector based on the lifetimes of objects". In: *Communications of the ACM* 26.6 (June 1983), pp. 419–429. DOI: 10.1145/358141.358147 (cit. on p. 34).

[66] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. "snmalloc: a message passing allocator". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management - ISMM 2019*. Phoenix, AZ, USA: ACM Press, 2019, pp. 122–135. DOI: 10.1145/3315573.3329980 (cit. on pp. 89, 93).

[67] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. "PAC it up: Towards Pointer Integrity using ARM Pointer Authentication". In: 28th USENIX Security Symposium (USENIX Security 19). 2019, pp. 177–194. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand (cit. on p. 21).

[68] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. URL: https://developer.arm.com/documentation/ddi0487/ha/ (cit. on p. 30).

[69] Arm Limited. *Arm Architecture Reference Manual Supplement Morello for A-profile Architecture version A.j*. URL: https://developer.arm.com/documentation/ddi0606/latest (cit. on pp. 10, 25, 26).

[70] Arm Limited. *Arm Morello Program*. Arm — The Architecture for the Digital World. URL: https://www.arm.com/why-arm/architecture/cpu/morello (visited on 01/02/2022) (cit. on pp. 10, 25).

[71] Arm Limited. *Arm Morello System Development Platform Technical Reference Manual* (cit. on p. 93).

[72] Arm Limited. *Arm Neoverse N1 System Development Platform*. URL: https://developer.arm.com/documentation/101489/0000 (cit. on p. 25).

[73] Arm Limited. *ARM7TDMI Technical Reference Manual r4p1*. The Thumb instruction set. URL: https://developer.arm.com/documentation/ddi0210/c/CACBCAAE (visited on 06/27/2022) (cit. on p. 26).

[74] Arm Limited. *BTI: Branch Target Identification*. ARM Developer. URL: https://developer.arm.com/docs/ddi0602/c/base-instructions-alphabetic-order/bti-branch-target-identification (visited on 02/06/2020) (cit. on p. 21).

[75] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification*. URL: https://docs.oracle.com/javase/specs/ (cit. on p. 27).

[76] Daiping Liu, Mingwei Zhang, and Haining Wang. "A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. CCS 2018. Toronto, Canada: ACM Press, 2018, pp. 1635–1648. DOI: 10.1145/3243734.3243826 (cit. on pp. 20, 69, 71).

[77] *Malicious malloc consumer can violate spatial safety · Issue #342 · CTSRD-CHERI/cheribsd*. GitHub. URL: https://github.com/CTSRD-CHERI/cheribsd/issues/342 (visited on 04/01/2022) (cit. on p. 88).

[78] Alfredo Mazzinghi. PhD thesis (cit. on p. 48).

[79] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4 (Apr. 1960), pp. 184–195. DOI: 10.1145/367177.367199 (cit. on p. 32).

[80] Myles McCormick, Derek Brower, Lauren Fedor, and Hannah Murphy. "Cyber attack sparks US effort to keep fuel lines open". In: *Financial Times* (May 10, 2021). URL: https://www.ft.com/content/b8b530c7-f194-43da-8c98-6e181f68da38 (cit. on p. 9).

[81] Kayvan Memarian, Victor B F Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N M Watson, and Peter Sewell. "Exploring C Semantics and Pointer Provenance". In: POPL 2019. Vol. 3, p. 32 (cit. on p. 16).

[82] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N M Watson, and Peter Sewell. "Into the Depths of C: Elaborating the De Facto Standards". In: PLDI 2016 (cit. on p. 16).

[83] Greg Morrisett, James Cheney, Dan Grossman, Michael Hicks, and Yanling Wang. "Cyclone: A safe dialect of C". In: UATC 2002 (cit. on pp. 16, 21).

[84] Santosh Nagarakatte, Jianzhou Zhao, Milo M K Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: PLDI 2009 (cit. on p. 19).

[85] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "CETS: compiler enforced temporal safety for C". In: *Proceedings of the 2010 international symposium on Memory management - ISMM '10*. ISMM 2010. Toronto, Ontario, Canada: ACM Press, 2010, p. 31. DOI: 10.1145/1806651.1806657 (cit. on p. 20).

[86] George C Necula, Scott McPeak, and Westley Weimer. "CCured: Type-Safe Retrofitting of Legacy Code". In: POPL 2002. Portland, OR USA, Jan. 2002 (cit. on p. 21).

[87] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: type-safe retrofitting of legacy software". In: *ACM Transactions on Programming Languages and Systems* 27.3 (May 1, 2005), pp. 477–526. DOI: 10.1145/1065887.1065892 (cit. on p. 21).

[88] Peter G. Neumann and Brett F. Gutstein. *CHERI's Mitigation of Hardware-Software Security Vulnerabilities*. Apr. 25, 2019 (cit. on p. 38).

[89] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. *Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches*. arXiv:1702.00719 [cs]. Feb. 2, 2017. URL: http://arxiv.org/abs/1702.00719 (visited on 05/10/2019) (cit. on p. 21).

[90] Aleph One. *Smashing The Stack For Fun And Profit*. Phrack Magazine. URL: http://phrack.org/issues/49/14.html (visited on 02/06/2020) (cit. on p. 17).

[91] *Oracle's SPARC T7 and SPARC M7 Server Architecture*. URL: https://www.oracle.com/assets/sparc-t7-m7-server-architecture-2702877.pdf (cit. on p. 21).

[92] Filip Pizlo. "Inline Caching in JavaScriptCore". URL: http://www.filpizlo.com/slides/pizlo-icooolps2018-inline-caches-slides.pdf (cit. on p. 30).

[93] Filip Pizlo. *Strings should not be allocated in a gigacage*. WebKit Bugzilla. URL: https://bugs.webkit.org/show_bug.cgi?id=185218 (visited on 02/22/2022) (cit. on p. 65).

[94] *Privilege escalation if a capability's bounds include the base address of a malloc'd block · Issue #1065 · CTSRD-CHERI/cheribsd*. GitHub. URL: https://github.com/CTSRD-CHERI/cheribsd/issues/1065 (visited on 04/01/2022) (cit. on p. 88).

[95] "Programming languages — C". In: *ISO/IEC 9899* (Oct. 2021), p. 638 (cit. on pp. 16, 70).

[96] Theo de Raadt. *Exploit Mitigation Techniques*. URL: http://www.openbsd.org/papers/ven05-deraadt/index.html (visited on 01/04/2022) (cit. on p. 17).

[97] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. "PACMAN: attacking ARM pointer authentication with speculative execution". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22: The 49th Annual International Symposium on Computer Architecture. New York New York: ACM, June 18, 2022, pp. 685–698. DOI: 10.1145/3470496.3527429 (cit. on p. 21).

[98] David D Redell. *Naming and Protection in Extendible Operating Systems*. 140. Massachusetts Institute of Technology, Nov. 1974, p. 169 (cit. on pp. 69, 71).

[99] Alexander Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. 949. University of Cambridge Computer Laboratory, June 2020, p. 189 (cit. on pp. 10, 23–25, 40, 41, 59, 60).

[100] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Transactions on Information and System Security* 15.1 (Mar. 1, 2012), pp. 1–34. DOI: 10.1145/2133375.2133377 (cit. on p. 18).

[101] Olatunji Ruwase and Monica S Lam. "A Practical Dynamic Buffer Overflow Detector". In: *Proceedings of the Network and Distributed Systems Security Symposium*. NDSS 2004. Feb. 2004, p. 11 (cit. on p. 19).

[102] Saelo. *Attacking Javascript Engines: A case study of JavaScriptCore and CVE-2016-4622*. URL: http://phrack.org/issues/70/3.html (visited on 02/10/2022) (cit. on p. 46).

[103] Qualcomm Product Security. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. Jan. 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf (cit. on pp. 21, 65).

[104] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *The 2012 USENIX Annual Technical Conference*. UATC 2012. Boston, MA, p. 10 (cit. on p. 21).

[105] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. "CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++". In: *Proceedings 2019 Network and Distributed System Security Symposium*. NDSS 2019. San Diego, CA: Internet Society, 2019. DOI: 10.14722/ndss.2019.23541 (cit. on p. 20).

[106] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. "SoK: Sanitizing for Security". In: *2019 IEEE Symposium on Security and Privacy*. SSP 2019. June 12, 2018. URL: http://arxiv.org/abs/1806.04355 (cit. on pp. 16, 21, 75).

[107] Eugene H. Spafford. "The internet worm program: an analysis". In: *ACM SIGCOMM Computer Communication Review* 19.1 (Jan. 3, 1989), pp. 17–57. DOI: 10.1145/66093.66095 (cit. on p. 16).

[108] *SPEC CPU 2006*. URL: https://www.spec.org/cpu2006/ (visited on 05/11/2022) (cit. on p. 93).

[109] Malcolm Stagg. *How I hacked my way to the top of DARPA's hardware bug bounty*. Medium. Jan. 30, 2022. URL: https://readme.security/how-i-hacked-my-way-to-the-top-of-darpas-hardware-bug-bounty-b66ec53b1973 (visited on 04/01/2022) (cit. on p. 88).

[110] *SunSpider 1.0.2 JavaScript Benchmark*. URL: https://webkit.org/perf/sunspider/sunspider.html (visited on 05/22/2022) (cit. on p. 59).

[111] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. SSP 2013. May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13 (cit. on pp. 38, 69, 71).

[112] Gil Tene, Balaji Iyengar, and Michael Wolf. "C4: the continuously concurrent compacting collector". In: *ISMM '11* (), p. 10 (cit. on pp. 34, 84, 97).

[113] *The Arm Morello Board*. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html (visited on 03/16/2022) (cit. on pp. 10, 25).

[114] *Twitter Investigation Report*. Department of Financial Services. URL: https://www.dfs.ny.gov/Twitter_Report (visited on 12/31/2021) (cit. on p. 9).

[115] US Department of Defense. "Department of Defense Trusted Computer System Evaluation Criteria". In: *The 'Orange Book' Series*. London: Palgrave Macmillan UK, 1985 (cit. on p. 36).

[116] *Valgrind Home*. URL: https://valgrind.org/ (visited on 03/11/2022) (cit. on p. 21).

[117] *WannaCry Ransomware Using SMB Vulnerability*. NHS Digital. URL: https://digital.nhs.uk/cyber-alerts/2017/cc-1411 (visited on 12/30/2021) (cit. on p. 9).

[118] Robert N M Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. 951. University of Cambridge Computer Laboratory, Oct. 2020, p. 590 (cit. on pp. 10, 22, 23, 25, 39, 72, 105).

[119] Robert N. M. Watson, Ben Laurie, and Alex Richardson. *Assessing the Viability of an Open- Source CHERI Desktop Software Ecosystem*. URL: https://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf (cit. on p. 38).

[120] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. Oakland 2015. San Jose, CA: IEEE, May 2015, pp. 20–37. DOI: 10.1109/SP.2015.9 (cit. on pp. 10, 22, 23).

[121] *WebKit JSCJSValue.h*. Feb. 1, 2022. URL: https://github.com/WebKit/WebKit/blob/1ecf88b40b4dce467f21b8cbc7d9a944f0a8fc74/Source/JavaScriptCore/runtime/JSCJSValue.h (visited on 02/01/2022) (cit. on p. 29).

[122] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. "Cornucopia: Temporal Safety for CHERI Heaps". In: *2020 IEEE Symposium on Security and Privacy (SP)*. SP 2020. San Francisco, CA, USA: IEEE, May 2020, pp. 608–625. DOI: 10.1109/SP40000.2020.00098 (cit. on p. 11).

[123] Davey Winder. *Uber Confirms Account Takeover Vulnerability Found By Forbes 30 Under 30 Honoree*. Forbes. URL: https://www.forbes.com/sites/daveywinder/2019/09/12/uber-confirms-account-takeover-vulnerability-found-by-forbes-30-under-30-honoree/ (visited on 12/31/2021) (cit. on p. 9).

[124] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. "CHERI Concentrate: Practical Compressed Capabilities". In: *IEEE Transactions on Computers* 68.10 (Oct. 1, 2019), pp. 1455–1469. DOI: 10.1109/TC.2019.2914037 (cit. on p. 22).

[125] Jonathan Woodruff, Robert N M Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *Proceedings of the 41st International Symposium on Computer Architecture*. ISCA 2014. Minneapolis, MN, USA, June 14, 2014 (cit. on pp. 10, 22).

[126] "Working Draft, Standard for Programming Language C++". In: *ISO/IEC 14882* (Nov. 2017). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf (cit. on pp. 16, 70).

[127] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. "CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 2019. Columbus OH USA: ACM, Oct. 12, 2019, pp. 545–557. DOI: 10.1145/3352460.3358288 (cit. on pp. 11, 69, 70, 73–75).

[128] Tadeu Zagallo. *A New Bytecode Format for JavaScriptCore*. WebKit. June 21, 2019. URL: https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/ (visited on 01/28/2022) (cit. on p. 28).