CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor

---

Lawrence G. Esswood

University of Cambridge
Computer Laboratory

Churchill College

July 2020

This thesis is submitted for the degree of Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Lawrence G. Esswood

July 2020

Abstract

CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor

Lawrence G. Esswood

This thesis presents the design, implementation, and evaluation of a novel capability operating system: CheriOS. The guiding motivation behind CheriOS is to provide strong security guarantees to programmers, even allowing them to continue to program in fast, but typically unsafe, languages such as C. Furthermore, it does this in the presence of an extremely strong adversarial model: in CheriOS, every compartment – and even the operating system itself – is considered actively malicious. Building on top of the architecturally enforced capabilities offered by the CHERI microprocessor, I show that only a few more capability types and enforcement checks are required to provide a strong compartmentalisation model that can facilitate mutual distrust. I implement these new primitives in software, in a new abstraction layer I dub the *nanokernel*. Among the new OS primitives I introduce are one for integrity and confidentiality called a *Reservation* (which allows allocating private memory without trusting the allocator), as well as another that can provide attestation about the state of the system, a *Foundation* (which provides a key to sign and protect capabilities based on a signature of the starting state of a program). I show that, using these new facilities, it is possible to design an operating system without having to trust the implementation is correct.

CheriOS is fundamentally fail-safe; there are no assumptions about the behaviour of the system, apart from the CHERI processor and the nanokernel, to be broken. Using CHERI and the new nanokernel primitives, programmers can expect full isolation at scopes ranging from a whole program to a single function, and not just with respect to other programs but the system itself. Programs compiled for and run on CheriOS offer full memory safety, both spatial and temporal, enforced control flow integrity between compartments and protection against common vulnerabilities such as buffer overflows, code injection and Return-Oriented-Programming attacks. I achieve this by designing a new CHERI-based ABI (Application Binary Interface) which includes a novel stack structure that offers temporal safety. I evaluate how practical the new designs are by prototyping them and offering a detailed performance evaluation. I also contrast with existing offerings from both industry and academia.

CHERI capabilities can be used to restrict access to system resources, such as memory, with the required dynamic checks being performed by hardware in parallel with normal operation. Using the accelerating features of CHERI, I show that many of the security guarantees that CheriOS offers can come at little to no cost. I present a novel and secure IO/IPC layer that allows secure marshalling of multiple data streams through mutually distrusting compartments, with fine-grained authenticated access control for end-points, and without either copying or encryption. For example, CheriOS can restrict its TCP stack from having access to packet contents, or restrict an open socket to ensure data sent on it to arrives at an endpoint signed as a TLS implementation. Even with added security requirements, CheriOS can perform well on real workloads. I showcase this by running a state-of-the-art webserver, NGINX, atop both CheriOS and FreeBSD and show improvements in performance ranging from 3x to 6x when running on a small-scale low-power FPGA implementation of CHERI-MIPS.

Acknowledgements

I would like to fill this space by thanking all the people whose talents and support made this possible.

First, my supervisor, Robert Watson, for the opportunity and freedom to pursue such an interesting project, and work with CHERI at all. Without your ideas and guidance, this work would look nothing like it does. You have been very patient.

Thanks to both of my examiners, Timothy Roscoe and Tim Harris, for all their help fine-tuning this thesis.

I am thankful to Arm for partially funding my time at the lab.

I am extremely thankful to Nathaniel Filardo, who has read and re-read this document. You have been an invaluable source of feedback, ideas, and interesting anecdotes.

Many thanks to David Chisnall, who taught me the basics of working with LLVM, and who I constantly interrupted seeking answers. I have learnt many things from you during my time at the lab, not least of which the importance of teatime.

Thank you to Peter Rugg, who scoured this document with me for errors. The fact we rarely find the same ones leaves me with a great deal of concern.

We truly do stand on the shoulders of giants, and none of this would have been possible without all the people who work on the larger CHERI project. You are too numerous to mention each by name, but I would like to give specific thanks to Alexander Richardson and Jonathan Woodruff, who always helped me with my compiler- and hardware-related difficulties.

I give my deepest thanks to all my friends (you know who you are) during these PhD years, new and old. I would not have got through it without you.

Last, but not least, to my parents, Aimée and Paul. Thank you for your years of compassionate and unwavering support. Although you may not understand it all, I hope this small success makes you proud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Security in contemporary computing

There is no doubt as to how critical computer systems are in today's world. We employ ever more computing resources across nearly every enterprise we undertake, now interconnected at an unprecedented scale. Computers control our banking, hospitals, and play a prominent role in many people's lives. As of 2019, over 3.3 billion people own an Internet-connected smartphone.[98] Although we have adopted computing technology at a neck-breaking pace, security has lagged far behind. The push to design cheaper, faster systems with security as an afterthought is commonplace, and the consequences of this behaviour are being felt. Widespread attacks with costly real-world consequences[7] have become regular news. In 2014, Heartbleed allowed arbitrary information disclosure on a sizeable fraction of websites.[46] Consequently, a US Hospital had 4.5 million customer records stolen.[15] In 2017, Wannacry ransom-ware infected hundreds of thousands of machines[47] causing widespread service outages across UK hospitals. In 2018, Spectre[73] and Meltdown[82] enabled disclosure attacks on the majority of contemporary Intel microprocessors. With the advent of cloud computing, businesses and individuals are concentrating and offloading our most sensitive data to untrusted third parties. The Internet-of-Things provides attack surfaces that cover a myriad of devices found in people's homes, and standards surrounding them are lax. Systematic vulnerability in our technology allows for attack automation, and ubiquitous networking allows their wide propagation. For example, Mirai[10] took just 4 months for 600 000 infections, targeting mostly IoT devices.

There are many case-specific ways a system could be considered insecure, so it is difficult to suggest a single technique solving all problem areas. Even so, the vast majority of attackers exploit the same class of simple and *avoidable* vulnerability now as they did decades ago. Memory-safety vulnerabilities are the root cause behind the majority of security bugs today, as much 70% in Microsoft products[26] and 58% in the Android kernel.[125]

Kernels, and operating systems (OSs) in general, are a crucial part of the contemporary computer. They are also particularly at fault when it comes to security vulnerabilities. Typically, OSs run with complete trust from their users, and complete authority over a computer. They are

large, complex in nature, and yet catastrophic to get wrong. Adversaries that compromise an OS subsequently win control over the entire system. Compounding this recipe for disaster, for speed and flexibility, languages like C have been and continue to dominate OS development. Although C offers great control over machine code produced, it is notoriously easy using existing toolchains to make mistakes that allow for system compromise. Languages like C, combined with unsafe toolchains, are predominately responsible for the memory-safety issues reported by Google and Microsoft.

Although crucial, achieving memory safety alone is insufficient. Not only do other avenues of attack exist, but further, not all vulnerabilities are bugs. In the days of more centralised computing, security was an issue of protecting a system from its users. As personal computing became ubiquitous, the framing shifted to protecting a user from multiple untrusted applications. Cloud computing represents a fusion of the two: a system hosts untrusted users, who in turn mutually distrust both the system provider and their applications. Even if all parties in a system could collaborate towards security, new conflicts of interest have developed such that trust between users and systems is no longer appropriate. Compartmentalisation allows isolating software written with different motivations, and as system software cannot be trusted, this requires enforcement by hardware. Intel's SGX[32] attempts to provide compartmentalisation with hardware enforced software enclaves. Due to its costs and inflexibility, use is rare even when supported. The threat of purposefully malicious OSs is real.[51][69] Even more troubling, authorities worldwide are making increased efforts to compel software vendors to backdoor their products.[110][122] Vendors do not have the correct incentives to protect their customers.

If we are to change this state of affairs disruptive change is required in how we build systems.

1.2 Breaking the hierarchy

Many techniques have been proposed and adopted that attempt to remedy memory-safety insecurities. Safe languages, automated fuzzing, and formal methods all provide some vulnerability reduction. Hampering efforts is inapplicability or unwillingness to use these methods, and a need for backwards compatibility. Furthermore, techniques that rely on being run atop a secure OS and runtime are fundamentally flawed. No such thing exists. Instead, we might consider it simply too difficult to write large-scale software without mistake[19], or indeed, security defects are purposefully introduced. Thus, we should design assuming the worst: OSs are adversaries. Compartmentalisation can reduce damage caused by vulnerable or malicious software components via isolation.

Existing OSs tend to use MMUs (memory management units) in conjunction with protection rings for security. MMUs can divide programs into separate virtual address spaces, such that programs cannot access memory belonging to either the OS or other programs. In turn, protection rings secure the mechanism by which the MMU is managed, making it unavailable to users. It is one of the fundamental roles of the OS to manage the MMU. Processes, an OS abstraction, marry an address space with some threads of execution. The granularity of compartmentalisation, in MMU-protected systems, tends to be the process, and trust is uni-directional.

Protection rings are hierarchical, intended to protect software running on lower rings from higher ones. Hardware enforces that running within an inner ring is required for certain privileged actions, such as would be required to manage an MMU. OSs achieve different privilege from users by running in a different ring. Hierarchies do not provide symmetric protection, so are insufficient for security purposes where mutual distrust between users and OSs is required. They also do not facilitate least privilege. Privileges required by system components are often too complex to be captured by a simple level, and the privileges required are not supersets of each other. A previous objective in microkernel design was to reduce the code running in lower rings to the smallest set possible. This makes the attack surface of the inner ring smaller, but does not address privilege reduction or mutual distrust. The problem is with the hierarchy itself. While a good structure for abstraction, it is a poor choice for security.

OSs require some privilege to operate, but here we de-conflate notions of trust and privilege. Rather than trust that privileges will not be abused, we assume they always will. A privilege powerful enough to break desired guarantees should not exist. This requires dispensing with a privilege hierarchy and replacing it with a flatter topology. From user programs to the microkernel, systems should consist of adjacent, mutually-distrusting, compartments. This model is similar to hardware-enforced enclaves, but taken to an extreme where every system component has its own enclave. Compartments can have the capability to perform privileged actions, but not necessarily a superset of another's. Every compartment is able to decide its own policy with respect to others. Software required to enforce policy always resides within a compartment. Failure to create a good policy only affects the faulty compartment. A misbehaving compartment reduces the usefulness of a system, but the system will always be fundamentally fail-safe with respect to properties such as integrity.

Not trusting the OS poses the problem of how to achieve any enforcement. Capability-based microprocessors, such as CHERI, can almost enforce enough to build an untrusted system, but fall short in places. CHERI provides protection primitives powerful enough for isolation, but cannot ensure that state is reached. I propose the introduction of a new hardware abstraction layer, the *nanokernel*, that introduces some safety checks and extra capability primitives but otherwise exposes hardware.

The nanokernel handles different abstractions than the microkernel. It only abstracts hardware features (such as interrupts and MMUs) so they cannot circumvent security. Unless required to enforce an invariant, the nanokernel does not sanitize access or provide high-level abstractions. It is still the microkernel's role to provide robustness, and to expose hardware in a uniform and shareable way. Consider, for example, the requirement of providing time-sharing. The microkernel decides what abstraction to provide, what their lifetimes are, when to switch between them, how isolated they should be, how communication is achieved, and where to save state. The nanokernel only helps with storing/restoring register files on request, and enforces that the microkernel cannot directly access memory used for storage. Most nanokernel state is exposed read-only to the microkernel, with modification allowed via simple mutators that perform sanitisation checks. With careful consideration of the primitives and invariants required, the complexity to implement a nanokernel is much smaller than for a microkernel. The CheriOS nanokernel is currently 2625 MIPS *instructions*. Compare this to seL4 (roughly 10 000 lines of C, ∼38 000

RISC-V instructions, or ∼40 000 AARCH64 instructions), or the MINIX microkernel (12 000 lines of C).[90]

This dissertation presents the design and implementation of a full OS running atop a nanokernel, dubbed 'CheriOS'. CheriOS is a single-address-space (SAS) OS, specifically designed for capability hardware such as CHERI. Unlike other SAS systems, the OS and all running programs are isolated from each other, even when malicious code is present. Although it only has one address space, CheriOS still uses an MMU for performance, but not for protection. CheriOS does not use protection rings, instead using only CHERI capabilities. On the CHERI-MIPS hardware in use, everything runs in kernel mode. The CheriOS microkernel exposes scheduler activations and a message-passing primitive. Other models, such as POSIX-like processes, are provided by system services and libraries. Management of the SAS is provided by a system service outside of the microkernel.

CheriOS tackles an adversarial form of memory safety, where other software components in the system – including those responsible for managing the MMU and scheduling – are actively trying to break memory safety. CheriOS shows how, even without trust, it is possible to design an efficient, easy-to-use system. Although CheriOS is a clean-slate OS not designed for compatibility, mostly unmodified C designed for UNIX can be compiled and run. CheriOS does not rely on any programmer effort to improve security for existing software. Entire classes of vulnerability, those at fault behind most contemporary attacks, are prevented. CheriOS does not necessarily do this with a performance cost. CHERI hardware-accelerates enforcement checks, and its costs are easily mitigated by removing the performance penalties traditional mechanisms already impose. CheriOS has a concrete implementation running on an FPGA, and was designed with larger-scale systems with virtual memory and multiple cores in mind. Although CheriOS is untrusted, other security measures are still employed as defence in depth. It is highly compartmentalised, and offers high-level guarantees of memory safety without trusting kernels, device drivers, or system services. Even the CheriOS microkernel tries to obey least privilege, having a minimal set of capabilities and the same ambient authority as other compartments.

1.3 Contributions

I propose the possibility, utilising CHERI, to significantly reduce the trusted computing base in a system. This is achievable by providing a few new capability primitives that facilitate least-privilege, untrusted OSs without committing to concrete OS design choices. I introduce the term 'nanokernel' for a security hypervisor that delivers these new primitives. Principally, the new primitives I introduce are:

Reservations Capabilities representing the right to allocate a specific range of the virtual address space. Use of reservations guarantees integrity and confidentiality for allocations, alongside guaranteeing memory safety in the presence of an MMU. Even if a reservation comes from an untrusted source, the memory that results from using it can be trusted to be private.

Foundations A set of capabilities allowing for attesting and reasoning about system state. Foundations allow verifying the authenticity of data in the system. They allow an abstract 'identity' of a program to act as a key for signing or protecting other capabilities. Foundations can verify which program produced a capability, or restrict which programs can receive capabilities, even if the OS message-sending primitive is untrusted.

I evaluate this design constructively, by prototyping a CHERI-MIPS nanokernel, and a message-passing single-address-space (SAS) OS atop it called CheriOS, which attempts to still be highly performant. Due to challenges encountered when designing a high-performance and untrusted system, I also devise and evaluate several new algorithms and data-structures, namely:

- A hardware-accelerated revocation mechanism to remove all copies and derivatives of a capability from a system.
- An IPC (inter process communication) API that utilises CHERI capabilities and a SAS to securely divorce control and data planes, while avoiding performance penalties that could result from increased compartmentalisation or reliance on encryption.
- A novel stack-allocation structure (slinky stacks), which the compiler employs to offer temporal safety.
- An ABI (application binary interface) that allows secure domain transitions between mutually-distrusting compartments using only CHERI primitives. This ABI includes the calling conventions and data structures used at a binary level to allow compartments to interface.
- A secure deduplication technique supporting system-wide, byte-granularity deduplication with complete distrust and without security loss. This serves not only as a replacement for MMU-based sharing, but also as a performance optimisation.

A major hurdle in adoption of security techniques is their performance impact, so I provide a detailed performance evaluation on the impact of my proposed methods, including their effect on a large piece of commercial software: NGINX. I do so by comparing performance with a contemporary OS, FreeBSD, ported to CHERI hardware. All benchmarks utilise a 64-bit MIPS implementation with CHERI extensions. It has a 5-stage, in-order pipeline, two cache levels (including a separate cache for capability tags), reminiscent of other lower-end RISC CPUs, such as an Arm7, but with a larger memory system. I also contrast my approach with other academic and industrial solutions.

An early CheriOS microkernel design was proposed by my supervisor, Robert N.M. Watson, and later implemented alongside some system software by Hadrien Barral. This previous work focused on how to build message-passing based IPC on a clean-slate OS using CHERI. My work inherits these ideas on microkernel design, but with a broader scope including memory safety, mutual distrust, and attestation. This earlier version has mostly been re-written due to

requirement changes, but the message-passing design remains. Some standard C library code was also imported from the FreeBSD project. NGINX, the LWIP networking stack, and the FAT filesystem used are all CheriOS ports of existing software. All other parts of the system, most importantly those contributing to security, are of my own design and implementation.

The CHERI architecture, the CHERI-MIPS FPGA implementation, and the LLVM-based CHERI compiler are the work of the CTSRD group at the University of Cambridge Department of Computer Science and Technology, and at SRI International. I make minor modification to the CHERI variants of the compiler tool-chain projects Clang, LLVM, and LLD, in order to provide a new ABI and support temporally safe stack allocation. I also modify both the QEMU-based CHERI simulator and the FPGA-based CHERI implementation to provide revocation mechanisms.

1.4 Thesis overview

Chapter 2 presents background material on capability systems and secure OS design. It includes a detailed description of CHERI, which is crucial to understanding CheriOS mechanisms. Chapter 3 gives an abstract, top-down overview of CheriOS, and serves as a brief introduction to the concepts that are further fleshed out in subsequent chapters. Chapter 4 covers memory safety. Chapter 5 shows how to build systems with mutual distrust. Chapter 6 demonstrates how we can achieve secure IO on compartmentalised systems without sacrificing performance. Each of Chapters 4 through 6 give a performance evaluation of any techniques introduced, comparing with others in the literature. Chapter 7 contains concluding thoughts about CheriOS and its future direction.

Chapter 2

Background

CheriOS is a capability operating system (OS) providing memory safety and compartmentalisation with low trust. This chapter contains a survey of background material surrounding system protection at a hardware, OS, and compiler level. It also provides an overview of capabilities, and how they have been used in OSs and hardware. The CHERI architecture is covered in more detail, as it is the basis for CheriOS.

Further details on some of the topics outlined here, and how they relate to CheriOS, are given in the related work sections 4.5, 5.5, and 6.4, after the relevant ideas in CheriOS have been introduced.

2.1 Software protection

Memory safety is a language-level property concerning access to in-memory objects. Although a high-level construct, it requires the assistance of hardware, OSs, runtimes, as well as correct language and compiler design. Even in unsafe languages like C, steps can be taken to increase safety.

We distinguish between two kinds of memory safety: *spatial* and *temporal*. Spatial safety forbids using a reference to one object to access others outside of its allocated bounds. Out-of-bounds array accesses violate spatial safety. Temporal safety concerns reuse of the same address for different objects over a machine's lifetime. Even with large amounts of memory, we are eventually forced to use the same memory for semantically different objects. Using an object after it has been freed and a new object is now in its place exemplifies violating temporal safety.

Safe languages

Safe languages are very popular at an application level. Java and C#, for example, consistently rank among the most-used languages.[21][87] They are designed to preclude memory-safety violations, normally incurring increased memory usage and decreased or inconsistent performance. In practice, they still suffer from memory-safety issues. Bad language design; buggy compilers,

standard libraries, and runtimes; or intrusion by other programs via system-level attacks can all violate the invariants that make safe languages safe. Each of these can be handled to an extent. Formal programming language semantics, alongside proof methods and enforced isolation, could allow us to trust our safe languages work. In reality, very few of these are ever achieved, let alone for an entire stack. The amount of code we rely on being correct for a program written in a safe language to truly be safe is huge. The Common Vulnerabilities and Exposures database[31] contains abundant results for 'Java SE' and '.NET Framework': 801 and 127 respectively.

The final issue with safe languages is the prohibitive cost of translating legacy software. Web browsers, mail readers, database-management software and many other complex yet safety-critical pieces of software have already been written in unsafe languages. OSs particularly represent huge investments. Replacing the likes of Windows, Linux, macOS and BSD with safer alternatives remains mostly the domain of research. For example, Mirage OS[84] is written entirely in OCaml, and Singularity[62] in Sing# (a C# derivative).

Formal verification

Mechanised proof engines are powerful tools for gaining confidence in the safety of large-scale codebases. Instead of requiring complete correctness of the code itself, we need only provide correct specifications in a formal language; automated tools can verify compliance and changes can be made until verification passes. This substantially improves on methodologies such as testing, which consistently misses bugs.

Formal proof is no magic bullet, however. It is possible to misstate specifications, or for them to be overly weak; a semantic gap will always exist between human intent and written specifications. It is also difficult to create proofs for real code. Formal proofs are often forced to exclude parts of the system where semantics are too inconvenient to capture. Furthermore, for proof to be possible, it is often necessary to avoid specific programming patterns.

In 2009, Klein et al. released seL4,[72] a formally-proven OS. However, it is limited by the aforementioned problems. To reduce complexity, the proof excludes multiple cores, many compiler optimisations, DMA, virtual memory, and assembly: all generally needed for modern high-performance systems. Specific program patterns, like taking references to local variables, were disallowed. Cost of proof is also high. The seL4 team estimates that while the kernel took only 2py to implement, it required 11py to prove it correct. They estimate trying to prove another similar kernel given their experience would cost 6py. Independent evaluators concurred that the cost of verifying projects of approximately 10 000s LOC was indeed feasible with this constant factor of development-time overhead.[9]

Although formal verification should be pivotal in designing safe systems, as current techniques do not appear to scale to large real-world software stacks (and have difficulty verifying arbitrary binaries), it seems unlikely to single-handedly solve all safety problems at every layer. It is currently best suited to verifying the small system components we are forced to trust, such as hardware and microkernels, using this trusted computing base as a foundation for a secure system.

Compiler techniques

Even for unsafe languages, compilers can insert instrumentation to perform dynamic checks that reduce or eliminate safety issues. The cost of these methods is normally prohibitive to deploying them for anything but debugging. For example, MemorySanitizer[124] incurs both memory and performance overheads between 2x and 8x merely to detect uninitialised memory usage.

One class of techniques, a type of blacklisting, stops accesses to unoccupied memory regions by blocking use of certain bytes. These stop overflows into unused memory, or use-after-free attacks if memory has not yet been reallocated. AddressSanitizer[117] does this in software, with a memory shadow-map, using one bit of shadow-space per byte of main memory. They report an average 1.73x performance overhead. The Califorms[111] work uses a novel encoding of the blacklist and hardware-accelerated checking. They report overheads between ∼1.02x and 1.16x. These techniques are, at best, probabilistic debugging tools, but are easily-circumventable as security mechanisms. Preventing corruption of heap metadata is a practical use of blacklisting. However, memory safety exploits do not generally access unused memory, they work by corrupting resident objects in occupied memory. Blacklisting bytes in-between objects can stop adjacent overflows, but not those targeting arbitrary offsets. Microsoft reports[26] adjacent overflows have dropped from comprising 90% to just 25% of detected memory-corruption attacks. Use-after-free attacks typically attempt to force reallocation with a specific object they wish to corrupt. This practice of 'spraying' heaps with valuable objects knowing they can later be corrupted is a standard trick in escaping browser sandboxes. Blacklisting does not stop this. The finest-grained blacklists also pose problems for intentional out-of-bounds accesses. Objects in an array may have bytes that should not normally be accessed, unless the array is being copied in its entirety. Skipping padding bytes while copying would make memcpy infeasibly complicated.

Bounds checking is an attractive countermeasure, but is also particularly expensive. Some methods prevent pointers ever going out-of-bounds using look-aside tables that record which memory region each pointer is allowed to point at. Jones et al.[68] use splay trees to look up bounds, reporting a median 6x slowdown. Others check accesses, but still use a look-aside structure for bounds. Softbound uses a two-tiered trie[94] for bounds lookup, reporting a 1.67x average overhead. Research on encoding bounds directly in pointers, so-called 'fat pointers', is also widespread. Cyclone[66] modifies the C language to feature them explicitly, at an average runtime cost of 1.6x, with outliers reaching 15x. Baggy bounds[4], whose compiler automates fat-pointer use, reports a similar 1.6x average slowdown on SPECINT 2000 benchmarks.

Tools also exist for temporal safety. Valgrind[97] can detect dangling-pointer dereferences provided the location is not reallocated, using a shadow-map of live allocations. It reports a 22x mean slowdown. CETS[93] uses a shadow-space (effectively a non-contiguous 196-bit fat pointer) to associate with each pointer a colour and another pointer to the colour that should currently be in use. There are sufficient colours to never require reuse. It reports a performance overhead of 1.52x, but incurs up to 3x memory overhead. Work on memory sanitisation is varied, but tends to recombine similar techniques. Song et al.[123] provide a good systematisation of knowledge of these approaches.

It is worth noting that although many memory-safety papers report average overheads in the 10s of percents, variance is often high. Benchmark suites such as SPEC are intended to illustrate a variety of workloads, not necessarily be a uniform sample of them. It is questionable to, for instance, run benchmarks that do not use malloc to justify a new malloc implementation is really fast on average. Use of geometric means is also common in memory-safety literature. Geometric means are always smaller than arithmetic means (for non-constant values), so papers that opt to use them can claim better results than those that do not. Even worse, for two distributions with the same arithmetic mean, the one with higher spread will have a lower geometric mean. Counter-intuitively, this means less consistent results look *better*. There is not much consistency in the mean used, and the choice is often unjustified or even unmentioned. The practice of reporting favourable averages in headlines, but leaving the truth deep in the paper that some benchmarks show overheads in the thousands of percents pervades. As examples, the ratios between the headline figure in abstracts and the worst-case overheads for Softbound[94], BOGO[100] and CPI[75] are 11, 28, and 9 respectively. There is no indication of such high potential overheads outside detailed results, which might convince readers that memory safety is mostly solved with only tolerable overhead. It is not, and memory safety issues still plague contemporary systems. This criticism is not specifically targeting these papers: their methodology is similar to others in the area and they were chosen arbitrarily. While averages are not useless metrics, papers are employing almost identical techniques yet claiming huge differences in results by averaging over unrelated workloads, or by choosing favourable interpretations of the word 'average'. I believe that, when employing benchmarks like SPEC, the range of overheads should be reported, as there is no sensible weighting usable to obtain a meaningful average. This may lead to less sensational headlines, but would avoid surprise for potential adopters.

There are also compiler techniques that attempt to statically verify memory safety, but, unlike formal methods, utilise heuristic measures to avoid manual effort. These can produce both false-positives and false-negatives. Coverity is a leading commercial tool for automated bug finding. An independent review of Coverity puts its false-positive rate at 25%.[5] False-negatives are much harder to quantify as it requires knowing exactly how many bugs a project has. Crucially, it is highly unlikely that the false-negative rate will be zero for a sufficiently complex project. For high degrees of assurance, heuristic-based tools are insufficient.

2.2 Hardware protection

Most application-class CPUs contain some form of hardware memory protection. Although they can enforce language-level memory safety, these mechanisms are generally only employed to isolate programs and the OS from other programs. Dividing and isolating software components is known as compartmentalisation, and aims to improve security and robustness for the overall software system. Most mechanisms are also completely and solely controlled by the OS, so cannot protect against an adversary who has OS control.

Segmentation

Segments provide both address-space virtualisation and access-control. A segment consists of a base, length, and permissions. Every access is made with respect to a segment at a given offset. When accessed, the MMU adds the segment's base to the offset, forming a physical address. It also compares the offset to the segment length, and access type with the permission bits. An access faults by being out-of-bounds, due to lack of permissions, or because the segment is not resident in memory. Typically, segments are used for each major program region, e.g. a segment for text, for data, for the stack, etc. However, one could also imagine a segment-per-object approach to provide safety. The B5000 is notable for doing this[78] for array objects. Section 2.3 expands on the B5000, and how its usage of segment descriptors relates to capabilities. The cost of a segment-per-object approach can be prohibitive, so segment protection tends to be coarse-grained.

x86 segmentation The meaning of segmentation on x86 processors differs based on the architecture version. Early iterations cannot properly be called segmentation, offering only base-address registers. Instructions could explicitly specify which segment register to use as a base. Segments lacked length and permission checks. Later iterations offered a 'protected' mode, where segment registers indexed into a descriptor table. Descriptors would contain not just a base but a limit and access permissions. Programs could not explicitly load descriptors as values, but rather had to use them indirectly via segment registers. The MMU on later iterations would read the descriptor table whenever a segment register was modified. To provide isolation, trusted code needed to manage the descriptor table correctly. Contemporary 64-bit x86 processors consider segmentation a legacy feature, preferring paged memory.

Paged virtual memory

Page-based virtual memory is common in contemporary application processors. Such systems split both virtual and physical memory into equally-sized *pages*. Like segmentation, pages virtualise memory and also enforce rudimentary access control. A structure called a pagetable maps from virtual pages to physical ones. When accessing virtual memory, the MMU translates a part of the address (the page number) from virtual to physical by look-up in a pagetable. Permission bits in page-table entries (PTEs) control both which operations are permitted, and the mode a processor has to be running in to use them.

A page-table walk per memory access would be prohibitively slow, so an associative cache called a *TLB* accelerates accesses. The TLB may be software-managed or hardware-managed, depending on the system. Most TLBs also support multiple address-space identifiers (ASIDs) for isolation, allowing processes in different spaces to access different pages without flushing the TLB.

Paged memory, when used for virtualisation, has many desirable features. Unlike with segmentation, consistent sizing means there is no external fragmentation. Used for access-control,

however, there exists drawbacks. Pages are relatively course-grained, a small page is generally 4KB. Inefficiencies are introduced by isolating using different address spaces. To share memory, multiple mappings need creating. Mappings take time to create, and extra TLB pressure further decreases performance. Furthermore, because pointers have different interpretations in different spaces, they cannot be transparently shared. This leads to extra copying and serialisation of objects, causing extra memory pressure and decreased performance.

Multiple virtual mappings can be used to control access permissions. For example, a virtual page can be made writeable (but not executable) for use by a JIT compiler to modify code. A different virtual page can be mapped to the same physical page as executable (but not writeable) for use by the application. The practice of ensuring no piece of virtual memory is both executable and writeable is known as W^X (Writeable exclusive or eXecutable), and inhibits attackers from injecting code into programs. This is standard practice on contemporary systems.

Rings

Hardware protection rings are a hierarchical set of architectural privilege levels. Generally, lower rings are more privileged and superset the privilege of higher rings. Rings protect particular functionality, including memory access and privileged instructions (e.g. those that modify exception vectors). Contemporary systems have multiple protection rings, generally including a ring for the hypervisor, kernel, and user.

Most OSs today (e.g. Windows and Linux), are monolithic, running a substantial amount of code, including third-party drivers, in a privileged ring. This risks any vulnerability in a substantial codebase enabling exploits that gain complete control over a ring and all those above it. Fundamentally, hierarchies of privilege hinder good compartmentalisation as they only allow nested compartments, not adjacent ones. Microkernels reduce the code running in higher-privilege rings, but still enjoy far more privilege than they really need.

Intel SGX

Intel's Software Guard eXtentions (SGX)[32] attempt to compartmentalise user applications from the rest of the system, including the OS. SGX allows designating a protected segment of memory as an *enclave*. The segment is initialised by the OS, but is encrypted by the CPU afterwards. A cryptographic hash of initial state is used for both local and remote attestation of enclave integrity. Enclave memory is only available when the CPU has a flag set by branching into the enclave at a pre-determined entry point. Enclave code can choose to voluntarily exit, and exit is forced on interrupt. Memory confidentiality is ensured by cryptography at the chip boundary, and a Merkle tree with its root on-chip provides integrity. If the OS reads enclave memory, it will see only the encrypted version (and it may legitimately do so to page out memory). If the OS corrupts enclave memory, either accidentally or deliberately, the Merkle tree will detect this fault and stop execution. Even a physical attack on RAM cannot deduce its contents, nor change the contents without detection.

Enclaves allow program compartments to have integrity and confidentiality with respect to the OS. They need only trust the minimal code shipped with the processor and the processor itself. SGX does have its downfalls. Like most methods discussed in this thesis, it is vulnerable to side-channels.[53][18][23][129] Furthermore, enclaves have substantial performance penalties and limitations on their size and number.[1] Performance penalties would increase with enclave size, and memory overheads are inherently required for security. It is currently infeasible to run entire VMs in enclaves, or run every process as an enclave. Enclaves also only provide a single level of protection; an enclave cannot contain a nested enclave. Details on enclave limitations are discussed further in section 5.5.

AMD SEV AMD's SEV[70] encompasses several technologies to provide hardware-enforced trusted execution environments. SEV protects guests from hosts on virtualised systems. Like SGX, SEV encrypts pages at the chip boundary. A different key is used per ASID, and only guests with the correct ASID see unencrypted memory. Unlike SGX, SEV lacks both integrity checks and nonces (IVs). This makes it more practical to employ at a VM granularity (there are no memory overheads), but has several security ramifications.

Lack of integrity allows replay and substitution attacks that leak VM contents by remapping pages.[91] The lack of nonces allows the host to observe if there are repeated plantexts. SEV also struggles with confidentiality due in its choice of encryption mode. The exact mode used is ECB (electronic code book), but plaintexts are first tweaked using physical addresses, avoiding the most trivial failure of ECB. The same plaintext in the same page will yield the same ciphertext. Du et al.[44] have reverse-engineered the exact tweak function and demonstrate arbitrary injection into the guest using a chosen-plaintext attack.

AMD have released a white paper on an upcoming extension[6] (SEV-SNP) that will add integrity to SEV by denying the host access to pages in-use by the guest. As the white paper is insufficiently detailed, it is unclear whether it fixes all issues in SEV. The same encryption mode is still being deployed.

Arm TrustZone

Arm's Trustzone[99] divides hardware resources into two worlds: a secure world and a normal world. A single mode bit, orthogonal to privilege ring, decides whether the processor is running in secure or non-secure mode. When in the non-secure mode, secure-world resources (such as certain memory locations and peripherals) are inaccessible. In the secure mode no such restrictions apply. A large, general-purpose OS can be run in the normal world, while a smaller, specialised one, alongside trusted applications, can be run in the secure world. Unlike SGX, this only allows the secure system to protect itself from the insecure. TrustZone aims to partition physical devices on a SoC rather than protect individual applications. TrustZone's implementation details are not generally publicly available, but attacks against it have still been demonstrated,[24] including those that are side channel based.[81]

[1] Intel's current implementation allows a total of 128MB of enclave memory.

Arm MTE

Arm's Memory Tagging Extension (MTE)[54] offers hardware-accelerated memory tagging at a relatively fine granularity. Both memory granules and pointers are given a colour. Hardware checks the colours match on dereference, or else throws an exception. Memory tagging is employed primarily to help enforce temporal safety. Much like compiler techniques that do the same thing purely in software, it relies on the runtime, allocator and OS to be correct in order to work. Due to limited space (4 bits) in pointers, colours must be re-used. ARM MTE must be combined with other methods, or only provide a probabilistic solution.

Intel MPX

Intel MPX adds additional pointer-bounds registers, and new instructions to create, load, store and check bounds. Like CETS[93] and SoftBound[94] (see 2.1), they use an out-of-band lookup structure. As bounds storage is out-of-band, existing programs should run unaffected. Even so, MPX breaks some existing C programs[100] (due to undefined behaviour), and is not thread-safe. It also has high (averaging 50%) performance costs and memory overheads, showing up to a 12x memory usage increase on one case study.[100]

BOGO[141] re-purposes MPX to provide temporal safety. When an object is freed, the bounds-lookup structure is scanned to find dangling references. BOGO sets bounds information for any dangling references to an invalid state. This is effectively a scan of every pointer on every call to free. This process is accelerated by only actively scanning a hot set of pages. Even so, it costs a further 50% average overhead, with some benchmarks being as much as 14x slower.

Using MPX is at the discretion of the compiler, and there is no run-time enforcement of it being used. Arbitrary bounds can be constructed without any privilege. MPX is usable by a program in protecting itself, but not in protecting from others. Since its introduction, MPX has been withdrawn from the instruction set.

2.3 Access control

This section covers mechanisms that restrict access to resources. We will use the terminology of *principals*, who operate on *objects*. To do so, they require *authority*.

Ambient authority

A principal may act under multiple authorities. Ambient authority refers to where principals only specify which objects to access, not under which authority to do so. Access control lists are an example of ambient authority. If a principal requests an action, the monitor allows it if any of the principal's authorities suffice. This may not be what the principal intends.

To illustrate why this is dangerous, consider the following real-world case, paraphrased from an article by Hardy.[59] There existed on a multi-user UNIX-like system a compiler. There was also a billing file ('BILL') recording who consumed system resources. Both BILL and the compiler could not be written directly by unprivileged users. The compiler could, however, write to BILL to audit its own use. When users ran the compiler they implicitly gave it their authority, and supplied an output file. The fault occurred when a user ran the compiler with an output file of BILL, as it would overwrite all billing information. The compiler acts under both its own authority and the user's. When opening the output file, the compiler did not specify an authority. It meant to use the user's, but instead used its own, something the designer of the program never envisioned. Nothing tied the name and authority supplied together, ensuring they be used in conjunction.

This division of object name and authority enables attacks known as confused-deputy attacks, coined by Hardy[59] for the previous example, where the confused deputy was the compiler. In software systems, principals are delegated authority from multiple sources and act on their behalf. Confused principals may accidentally use a more privileged authority than intended. If prompted by an attacker with little privilege of their own, the result is privilege amplification.

In hardware, rings are an example of ambient authority. Occupying a lower ring implies privilege. Similarly, pagetables, segment tables, TrustZone's secure world, and SGX enclaves are all cases of ambient authority. All security checks are performed implicitly on memory access. Systems built atop them need not feature ambient authority, see section 2.3, but underlying hardware is prone to ambient authority problems.

Capabilities

A capability[39][89] is an unforgeable token that both names an object and confers the authority to the bearer to act upon it. Capabilities can be delegated between principals to both identify an object, and simultaneously grant some authority over it. Contrast this with other methods that use two channels to share objects, one for the name, the other for authority. For example, on UNIX, a file name is transferred as a plain-string path, and the authority to open it is transferred by separately modifying permissions. A capability-fied version would be a 'path capability', which would not just represent a path but the authority to open it in specified modes.

Every access to an object requires authorisation by a capability. It is impossible to confuse which authority should be used for a particular operation as it forms part of the reference. The capability used is not always explicit. Some machines, like the CAP, perform dynamic capability lookup. The CHERI[37] work defines the *principle of intentional access* as where, if a set of authorities is available, the invoked authority is stated explicitly. Systems that are not explicit in which capability to use, or eschew capabilities completely, are vulnerable to confused-deputy attacks. Capability systems that obey the principle (such as CHERI and CheriOS) are not.

Capabilities also separate policy from enforcement, reducing complexity in parts of the system we must trust. Capabilities can be shared in a decentralised way, determined by principals. As a result, capabilities tend to proliferate throughout a system. Revoking a capability is to

globally destroy or invalidate it and potentially any capability derived from it. There is normally a trade-off in complexity of use, and complexity of revocation. Some schemes[107] will use tracking structures that require upkeep, but facilitate revocation. CHERI requires no such tracking, favouring a more complex revocation procedure to accelerate capability usage.

Despite many notable examples,[78] most commercial systems today are not capability-based. Modern filesystems, program permissions, and memory protection features all, for the most part, use ambient authority.

Capability machines

Capability machines provide and enforce capabilities in hardware. This is advantageous not just for performance, but also for reducing the trusted computing base. Although currently unpopular, there have existed numerous capability machines, both academically and industrially. The 1960s Burroughs B5000[88] stack machine had a descriptor table for every program segment. An array access would first calculate an index in the descriptor table and then indirect through it. Descriptors had bounds, and hardware performed access checks. Although this preceded the modern notion of a capability machine, the Burroughs B5000 was the first to utilise tagged memory to distinguish between segment descriptors (notionally capabilities) and other memory types. This allowed entries to be loaded directly onto the stack.

The 1970 Cambridge CAP computer[96] was another early register-based capability machine. The CAP computer used explicit capability registers, which it could load and store from dedicated capability segments. Dedicated capability segments obviated tags, but prevented mixing capabilities and data flexibly.

Commercially, IBM's System/38,[61] and its successor the AS/400, were successes, and like the B5000, featured tagged memory to distinguish capabilities from data. Intel's iAPX 432 was less successful due to poor performance, notably due to its poorly-optimised ADA compiler.[29] It supported both hardware- and software-defined capabilities, differentiating between data and capabilities like the CAP computer, by dividing segments.

The M-Machine[50] used guarded pointers for capabilities, as described by Carter et al.[20] It supported power-of-two sized bounds and bound alignments, using a fat pointer to encode bounds, permissions, and an address. M-Machine capabilities supported two interesting capability variants: a 'key' capability and an 'entry' capability. Neither could be modified or forged. Entry capabilities could be jumped to, producing executable capabilities, and key capabilities served as identifiers. A bit in executable capabilities encoded whether the processor was in supervisor mode. Combined with entry capabilities, this allowed securely transitioning to privileged code at pre-defined entry points using normal jumps.

More recently, in 2013, LowFat[76] implemented hardware support for bounds checking fat pointers compressed into 64 bits, with support for bounds-alignment not on a power-of-two. They are, although similar to M-Machine and CHERI constructs, not capabilities. They lack authority, unforgeability, and a facility for compartmentalisation.

CHERI[136][135] was designed as part of the CTSRD project. It further refines hardware-enforced capabilities, but with a goal shift to run existing MMU-based OSs, and mostly unmodified legacy C/C++ code. Unlike M-Machine, CHERI avoids software-emulation for capability operations, but still incorporates many of its ideas. Due to its importance in understanding CheriOS, CHERI is described in detail in section 2.4.

Capability operating systems

Just as capabilities are useful for hardware-level access control, they find use at an operating-system and application level. Although capability hardware like the CAP also had capability-based operating systems, it is not necessary to have a capability machine to construct software-defined capabilities. There are many academic and in-production examples of capability OSs. Provided here is a brief overview. More details on some of these systems (namely Eros, seL4, and the CAP OS) and how they relate to CheriOS, are given in section 5.5, once the main ideas of CheriOS have been introduced.

An early system, Hydra,[137] was built atop the PDP-11. Hydra had capabilities for system-level objects and user-constructed objects. All capability operations required syscalls as only the kernel could be trusted not to forge capabilities. This limited the frequency of operations and which objects could practically be represented by capabilities. However, Hydra demonstrated implementation of the object-capability model via IPC.

KeyKOS (originally GNOSIS) was another early commercial capability OS, first developed for the IBM System/370, and could support (among others) a UNIX environment.[60][17] KeyKOS, like most of the capability OSs discussed here, was a message passing system, and used capabilities (which it called keys) to represent most system resources. KeyKOS separated the system into domains, and gate keys could be invoked to send messages from one domain to another. KeyKOS implemented a single-level persistent store. This store could be manipulated from userspace using capabilities, and was transparently swapped with backing storage by the kernel. KeyKOS was followed by Eros,[118] and later Coyotos,[34] which are both research focussed systems.

The Mach microkernel[106][1] offered *tasks*, a collection of threads combined with a virtual address space. The Mach abstraction for IPC was a *port*, which was much like a KeyKOS gate key. The authority to communicate on a port was represented by a capability, and capabilities themselves could be sent over ports for delegation. Messages, sent synchronously or asynchronously, were dynamically-sized and typed. Like Hydra, Mach implemented an object-capability model using IPC. For example, creating a thread required sending a message on a specific port, and the authority delegated by sending the port capability. Mach aimed to provide a kernel compatible with a UNIX environment,[52] while hosting maximum functionality outside of the kernel as a set of servers. Multiple UNIX variants have been derived from Mach, or used Mach as a microkernel.

The L4 microkernel was designed after Mach: its goal was to further reduce complexity and overheads compared with past microkernels. It featured a simplified interface (an early version featured only 7 system calls[79]), offering only threads, address spaces, and IPC. Simplified IPC meant that across-address-space message send on the first L4 implementation was 20x[79]

faster than Mach. SeL4,[72] as discussed in section 2.1, is a formally-verified, from-scratch implementation of L4. SeL4 stores capabilities in a tree, where each node is an array of capabilities. Capabilities are fat-pointers to objects in the physical memory space, begin untyped, and are re-typed to construct objects, such as page-tables. Capabilities are only modifiable by the kernel, and are referenced from user-space via index.

Barrelfish is an OS developed by ETH Zurich,[113] designed to better support large-scale heterogeneous systems. Barrelfish uses capabilities[119] similar to seL4's, with extensions to deal with the increased complexity of deleting, copying, re-typing and revoking in a distributed fashion. Like other capability-based systems, capabilities represent physical memory, kernel objects and communication end-points.

Capsicum,[132] extends the POSIX API with file capabilities, and was prototyped on FreeBSD. Capability operations still require system calls as FreeBSD runs on conventional hardware. Although files represent many things in UNIX, Capsicum does not provide memory capabilities. Therefore, pointers shared between the OS and userspace can be arbitrarily fabricated.

CheriBSD[37] augments FreeBSD to utilise CHERI hardware-enforced capabilities. CHERI capabilities can protect memory accesses both in userspace compartments and the kernel. Being based on FreeBSD, much of the system interface is not capability based. For example, spawning threads on FreeBSD is done via syscall, with ambient authority to do so. Mach would use a capability for such an authority. While other capability OSs also have memory capabilities, their use is more coarse-grained than CheriBSD's. An OS like seL4 uses memory capabilities to, for example, grant the authority to create a page mapping. The *use* of that mapping, via a load or store, will not check a capability. Also, because systems like seL4 use the MMU for enforcement, sizes and alignments of objects capabilities refer to are necessarily quite coarse.

2.4 CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) extends existing ISAs with hardware-defined capabilities. CHERI aids adoption by being backwards compatible with existing software. The most mature instantiation of CHERI is currently CHERI-MIPS, having both a simulator and FGPA implementation. Disambiguating CHERI and CHERI-MIPS is unimportant for the purposes of this document, apart from noting that although CheriOS was developed atop CHERI-MIPS, it is unreliant on MIPS-specific features. Both CHERI-RISCV and CHERI-ARM (codenamed Morello[92][55]) are under development.

CHERI capabilities

CHERI capabilities, like M-Machine's, are fixed-length register-sized values allowed to reside anywhere in memory. Tagged memory protects the validity of capabilities and stops arbitrary modification. Software cannot set the tag bit explicitly. On system start, there is only a single

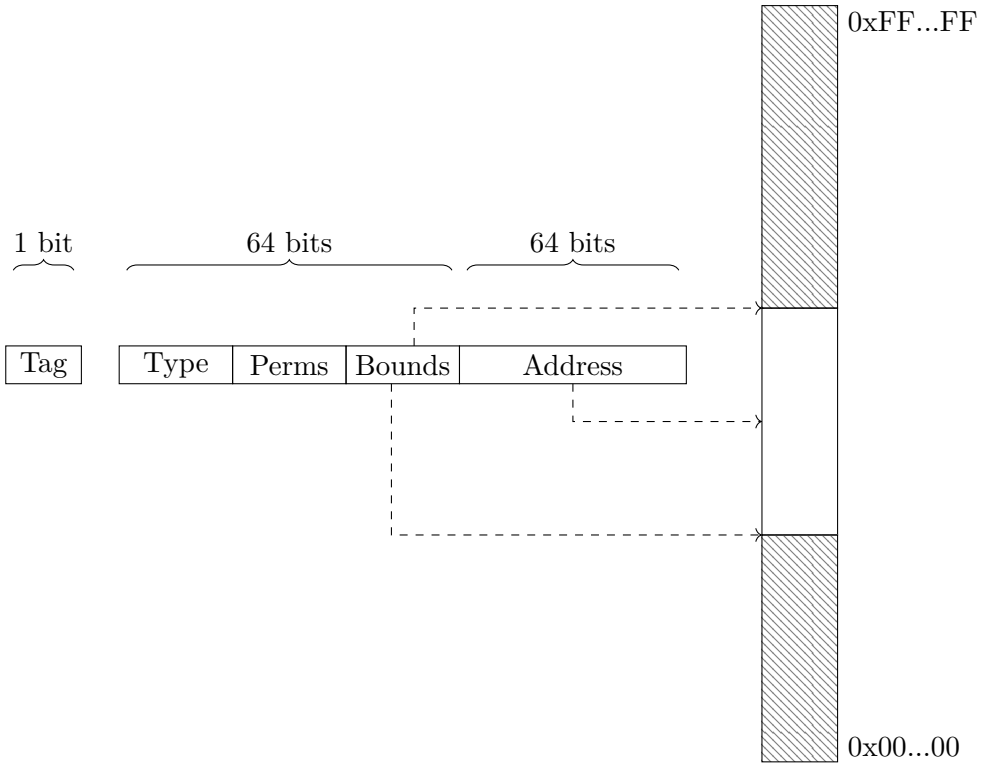


Figure 2.1: *A CHERI capability*

all-powerful capability available, the *primordial capability*. New capabilities are only obtainable via monotonic operations on existing capabilities. That is, any capability manipulation results in a capability that is less than or equal to (in privilege) its input. Even untrusted or privileged code cannot construct capabilities from nothing, or amplify one. Programs can restrict a capability to produce a new one, e.g., by removing permissions or shrinking bounds. The bootloader will bootstrap the system using the primordial capability. All capabilities are derived from it; losing the last copy of a capability would result in having no way to regain access, short of reset.

Different iterations of CHERI have different capability formats, the current MIPS format is shown in figure 2.1 and is 16 bytes (without the tag), with an 8-byte address space. A capability has an address, bounds, permission set, type, and tag. Although the tag can be read from a register, it cannot be written and is not visible in memory, and so might be thought of as separate. Sub-capability sized writes clear the tag. All memory accesses must use a capability with bounds covering the desired location, with the correct set of permissions for the access. Capabilities can be used in place of pointers in programs, and in CheriOS this is always the case. Although both upper and lower bounds are 64-bit addresses, they are compressed in memory. An artefact of this is alignment requirements on bounds. CHERI bounds encoding is more complex than that of M-Machine and Low-Fat, allowing greater out-of-bounds representability and fewer requirements on alignment.

Although capabilities have an address and bounds, not every capability should be considered a capability to memory. The architecture defines more actions than memory access that capabilities

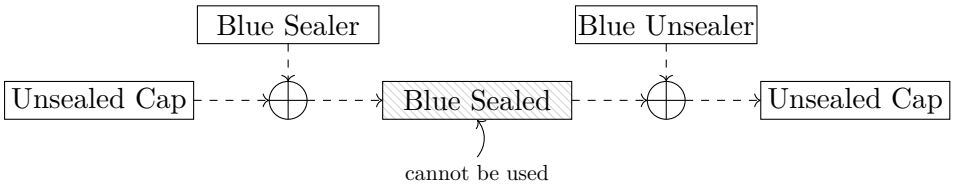


Figure 2.2: *Sealing mechanism. An unsealed capability is sealed with type blue, then unsealed again to produce the original capability.*

can authorise. Furthermore, software defines the meaning of sealed capabilities, see section 2.4. We will refer to capabilities with memory access permissions as *memory capabilities*. Executable capabilities we may refer to as *code capabilities*, or else as *data capabilities*. We call capabilities with sealing permissions *sealing capabilities*, and those that allow certain architectural operations, such as installing TLB entries, *architectural capabilities*.

Architectural capabilities

Memory access is not the only mechanism that needs protecting. Privileged instructions that modify control status registers (CSRs) need to have their use restricted. TLB management effects the interpretation of memory capabilities, so if unprotected could subvert protection. CHERI does not specify different permissions for every architectural operation. Instead, CHERI groups all such permissions into a single 'Access System Registers' bit in the capability format. A capability with this bit is used by installing it in a register: on MIPS this must be the program counter (PCC). Architectural capabilities allow fine-grained control of code allowed to execute privileged instructions. Historically, we relied on privilege rings for this, which allow far less control. Using architectural capabilities, unprivileged code can be run side-by-side with privileged code in the same ring.

Using PCC to hold architectural capabilities conflates code capabilities with architectural capabilities. In a purer world, we would have a separate architectural capability as an explicit argument to privileged instructions, but a single capability integrates better with existing kernels. Importantly, use of PCC is still intentional use. There is no ambiguity which capability authorises an operation. PCC should be viewed as an implicit argument to all privileged instructions, similar as to how outputs of arithmetic functions might have to be a specific register. There is no risk of accidental transfer of privilege when calling a function, as privilege bits cannot be transferred between capabilities.

Sealing

Capabilities are used for more than authorising architectural operations. Capability OSs use capabilities to represent abstractions like files. The meaning of these capabilities is software defined, but enforcement and unforgeability is still provided by hardware. This is a prime example of how capabilities can separate enforcement from policy. The CHERI mechanism for software-defined capabilities is called *sealing*. A sealed capability is like an M-machine 'key' / 'enter' capability, or a hydra 'restricted' capability.

A sealed capability cannot be used or modified, apart from to be unsealed. See figure 2.2 for a depiction of sealing. The authority to seal or unseal a capability is granted by another capability with the 'seal' or 'unseal' permission bits, respectively. The address and bounds of *sealing capabilities* refer to a type space rather than a memory space. When capability X seals capability Y, capability Y's type will be the address of X, provided it fits within the type field. In order to unseal Y, an in-bounds capability with the same address of X and the 'unseal' permission is needed. Although values and bounds of both sealing capabilities and memory capabilities are represented in the same way, memory location 0x1000 is not the same as type 0x1000. A capability could have both memory and sealing permission bits, but as this conflates the two spaces it is not done.

Sealed capabilities are used in two ways: as bearer tokens, or as a means to support the object-capability model.

Bearer tokens

The first use of a sealing capability is to construct bearer tokens. A bearer token is a capability that the bearer will pass when performing an operation to authorise it. Whereas other capabilities have hardware-interpreted meaning when used by instructions, sealed capabilities used as bearer tokens are interpreted by software. Sealing can be used to safely pass object handles to untrusted parties, without granting any authority to architecturally manipulate the object. The sealed capability is still unforgeable, and can only be constructed by a principal with both the unsealed capability and the relevant sealing capability.

For example, a filesystem could seal capabilities to memory records representing file handles. When unsealed, the capability is an architecturally-defined memory capability to filesystem state. When sealed, a reference is interpreted as a file handle. This is particularly useful as the mapping between handles and the filesystem's internal state is performed using a single unseal instruction. Users cannot dereference the sealed capability, nor can they forge their own handles.

There is no need to seal a capability that has any architecturally-defined authority. If integer handles were desired (as is the case in UNIX), a tagged capability that has no permissions could be sealed to make these integers unforgeable by users. The unsealed capability confers no authority, only the sealed version has meaning, and only to the principal that sealed it for use as a bearer token. Capabilities with no permissions are considered to cover a third integer space, alongside the memory and type space. A software defined capability can represent an arbitrarily complex object, even though hardware only supports a few primitive operations. Constructing them is a cheap, pipeline-able instruction that does not access memory.

Object capabilities & CCall

The second use of sealing, in conjunction with the CCall instruction, is to provide object capabilities. In the object-capability model, a capability represents the invocable authority to perform an operation on an object. It provides an enforced model of encapsulation.

Sealing and unsealing capabilities on its own does not provide a way to transition between security domains. Traditional systems use synchronous exceptions to simultaneously transfer control-flow and also change protection ring. This is expensive, inflexible, and is unsuited for fine-grained transitions between multiple domains. CHERI features an exception-less 'CCall' instruction, that achieves a similar transition using CHERI capabilities.

CCall takes two sealed capabilities: a code capability, and a data capability. The code and data capabilities require a permission bit that makes them eligible for CCall, and must also be sealed with the same type, or both must be unsealed[2]. CCall will unseal and jump to the code capability, whilst also unsealing the data capability into a general purpose register called IDC. Together, the two capabilities used with CCall are an object capability, and CCall is the instruction to invoke it. Those familiar with Hydra might consider CCall to be a hardware-implemented procedure call, with a template consisting of a single parameter requiring a 'CCall' right.

CCall allows unsealing data capabilities without a corresponding unsealing capability, but only by transitioning to a code capability that was constructed by the type owner. Like the exception mechanism, this transfers control to trusted code whilst also elevating privilege (by unsealing the data capability). The code capability may have the 'access system registers' privilege, so CCall can be used to transition to code that uses privileged instructions. CCall has practically the same cost as a jump, can make any memory location a domain entry point, and can pass multiple capabilities between caller and callee, either directly via argument registers, or indirectly by reference. A single data argument being unsealed by hardware is not a limitation. The capability unsealed into IDC can load subsequent capabilities from the new domain, or even a capability to perform unsealing of further arguments. Importantly, CCall is unreliant on trusted third-parties (apart from hardware) to achieve a transition. This is fundamental in using it to create mutually-distrusting compartments.

2.5 LLVM

LLVM[83] is a large, open-source project consisting of a suite of compiler-tool-chain sub-projects, with active research and development from both industry and academia. LLVM also refers to the compiler 'middle-end', a source- and target-independent optimising compiler that started as research project with the same name.[77] LLVM operates on LLVM-IR, a typed language for a virtual register machine, more low-level than most programming languages, but generic enough to map to multiple architectures. LLVM is designed for modularity and extensibility. Programmers can insert passes that manipulate LLVM IR at a basic-block, function, or compilation-unit granularity. These passes can be inserted anywhere in the LLVM pipeline to add extra optimisations or analyses.

LLVM back-ends exist to lower LLVM-IR to target-specific instructions for multiple mainstream architectures, including MIPS. Clang is an LLVM front-end that can lower C and C++

[2]CCall, as described by CHERI, does not allow the unsealed case. I have added this as CCall variant 2.

to LLVM IR. LLD is the LLVM-project linker, intended as a drop-in replacement for the UNIX LD utility.

Previous and ongoing work on the CHERI[136][37] project have made extensive modification to Clang, LLVM, the MIPS back-end, and LLD, in order to support C programs on CHERI machines. CheriOS utilises this effort, with further extensions to support the CheriOS calling conventions in section 5.1, and memory-safe stacks as described in section 4.1.

Chapter 3

Designing CheriOS

This chapter gives a top-down overview of CheriOS and some motivation behind its design. It provides some context and a brief feature overview to aid the understanding of other chapters, which present specific details.

Operating systems manage and mediate access to hardware resources such as CPUs, memory, and peripheral devices. They will often offer high-level abstractions, such as filesystems, that hide the complexity and heterogeneity of underlying hardware, and implement commonly required functionality. OSs also enforce some degree of isolation and resource arbitration, so applications cannot compromise each other or the OS. The core component of an OS is its kernel, which will typically run with a high degree of trust (such as being run in a privileged ring), and is often less fault-tolerant than other system components.

A monolithic kernel contains all OS services, such as filesystems and device drivers. Alternatively, microkernel OSs (such as most discussed in section 2.3) obey the minimality principle, which dictates only strictly necessary functionality be included. Typically this is taken to mean at least some memory management, time sharing, and IPC. Microkernels offer benefits of portability, maintainability, and crucially for systems that desire it, security.

The hope when designing security-oriented microkernels is that an attacker cannot compromise the microkernel, as the attack surface is sufficiently small. This is not true in practice. Microkernels, although small, are still complex enough to expect bugs, absent of formal verification. Formally proving something, even as small as a microkernel, is difficult.[72] In the presence of an attacker who gains code execution within a microkernel, minimality achieves little: all that matters are the powers available to the microkernel, even if never normally leveraged.

Minimality crucially differs from the security principle of least privilege. Least privilege requires giving entities only the authority required for their goals; minimality only requires avoiding use of authority in overly-complex ways. Conventional microkernel designs fall short in this regard. Rings are, by nature, antithetical to least privilege. Inner rings grant strictly greater authority, rather than fine-grained sets as would be required for good least-privilege design. There

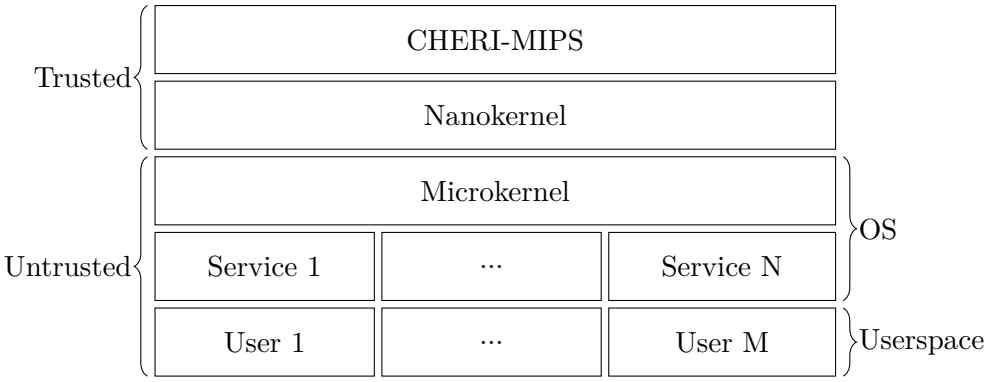


Figure 3.1: *CheriOS software stack*

is little merit in restricting the roles a microkernel plays if it still holds all the authority to play any role when compromised.

CheriOS attempts a microkernel with minimality and least privilege. However, it is infeasible for a microkernel to truly obey least privilege, given the interface a modern ISA provides. For example, microkernels require the authority to suspend tasks, but probably not to inspect their state when doing so. Interrupt-delivery primitives offered by conventional ISAs cannot provide one without the other. Even with CHERI, architectural capabilities are too few (CHERI has one) and too powerful to split among compartments. Even if CHERI divided its architectural capability, such as to provide an architectural capability to manage interrupts, this would still be too powerful for least privilege.

To remedy this, we introduce a new security hypervisor atop the ISA dubbed the *nanokernel*. The nanokernel is the only software component in the system that applications must trust in order to guarantee isolation and secure communication. Along with the hardware, it is only system component in the TCB, where correctness is assumed when making claims about any security guarantees offered by CheriOS. Other software can always be untrusted, so only usefulness is sacrificed on failure.

Some software that (potentially) runs outside the system must also be trusted. The compiler is still a part of the TCB; safety relies on compiler generated code behaving as specified by the program. The techniques developed in this thesis are enabled by default when compiling programs for CheriOS. Importantly, use of a malicious or buggy compiler by other components (including the OS) can only impact their own security. This means that the provenance of programs need not be known, and programmers need only ensure they use the correct compiler for their own programs. This is unlike entirely compiler-enforced mechanisms that rely on every component using the correct compiler.

The CheriOS software stack is illustrated in figure 3.1. There is exactly one instance of hardware and the nanokernel, and all software interacts with both. The nanokernel is agnostic to the exact structure of other software, but CheriOS follows the design of previous microkernel-based OSs: a microkernel (implementing abstractions like tasks and IPC), some system services layered over the microkernel (providing other abstractions and common functionality), and several

user applications atop both. It is important to note that this hierarchy is only interactional in nature. The hierarchy of trust follows a different structure, and is flatter. That is, every software component trusts hardware and the nanokernel, and the nanokernel trusts the hardware. Other trust relationships are completely flat: every component distrusts every other and can independently ensure their own isolation using nanokernel primitives (although they can choose to opt out).

3.1 System model and security guarantees

CheriOS tries to map language- and system-level models (such as objects and functions) down into hardware in such a way that the higher-level semantics are also enforced by hardware. This allows certain guarantees without trusted compilation, or trust of most of the code running on the system. In 3.1 we give a description of the CheriOS model, whose security guarantees stem from its enforcement. Capabilities to system-level objects are always represented by hardware capabilities. The mapping from language-level objects to objects at a hardware-level is somewhat compiler-defined, but is designed to be one-to-one. The mapping used by parts of the system responsible for loading programs, and by the default CheriOS compiler, are outlined in 3.1.

System model

CheriOS is composed of a single space of fixed-sized, non-overlapping, completely independent memory objects (modifying one cannot change another, nor does reading one give the contents of another). These objects can include any mix of instructions, plain data, or capabilities. CheriOS guarantees the following about objects:

- An object can only be read or written by invoking a capability to it, and a given capability never changes what object it refers to.
- A capability cannot be forged and can only be derived from an existing capability with equal or greater authority to access the same object.
- When an object is created, the capability to it is unique.

The objects in CheriOS form a graph, where nodes are objects, and edges the capabilities they contain. The system is framed as a collection of domains. A domain is a set of reachable capabilities, identified by some root set of capabilities, and transitively closing that set under accessibility. Domains can contain sealed capabilities which are entry points to other domains. Invoking a sealed capability will gain a new root set of capabilities, transfer flow into some pre-arranged instructions in that domain, and optionally pass some capabilities as arguments. CheriOS guarantees the following about domains:

- If two graphs are disjoint, those domains are isolated, or if they overlap, the domains only share those objects which are reachable from both (possibly with differing access rights). In this way, degrees of isolation are flexible.
- The behaviour of a domain is completely defined by the code within it.
- Only capabilities that are intentionally released by a domain are transferred when switching between domains.

As long as malicious code is constrained to only other other domains (not including the nanokernel), an isolated domain is guaranteed integrity and confidentially for both its objects any control-flow within. No mismanagement of hardware by the rest of the operating system is allowed to break the guarantees concerning domains or the objects they contain.

Mapping into languages

The above model can be mapped neatly to higher-level notions of processes and programming languages. Objects, as they exist in languages like C, are one-to-one with the objects as described above. The data structures that exist at a language-level map to the capability graph that exists at a hardware level. This ensures C programs will be memory safe, as is detailed in section 4. A CheriOS process will typically be built out of multiple domains. By default, a CheriOS process will ensure (without trusting the OS to construct them in such a fashion) that the domains it is composed of are completely disjoint from the rest of the OS. Furthermore, processes can be internally compartmentalised, ensuring that the state of each thread, in each compartment, is a separate domain (sharing only programmer defined global state). These compartments can be as large as libraries, or as small as single functions. Language-level function calls map naturally to entering domains, where the capabilities passed to the domain are the arguments to the function, enforcing that only intentionally passed objects ever flow between functions, which is an enforcement of scoping rules of the language.

The guarantees offered by CheriOS are somewhat weaker than that offered by trusted compilation and a safe language and runtime. Some properties, such as type safety, are not enforced at all (although there is nothing stopping a programmer using a language and compiler on CheriOS that does have such guarantees). Rules of scoping and control flow are only strictly enforced at domain boundaries. Depending on how coarsely these are placed, some accesses not allowed by source code could occur at runtime. Some degree of protection is still offered to protect a domain from itself. Memory safety, for example, is always enforced even within a single domain, as long as the default compiler was used for code in that domain. Protection is also more localised. A given domain cannot ensure that other domains are interacting in a safe fashion. For example, a domain can only ensure control flow into and out of it is protected, and not that control flow between other pairs of domains is legal. It is the default assumption that each domain behaves as if every other domain were malicious, and cooperating with each other against it. Finally, CheriOS offers no guarantees of progress. Mismanagement of hardware by CheriOS can cause progress within a domain to irrevocably stall, either by unmapping resources, withdrawing services, or just not scheduling anything. Although weaker than the protection

offered by verified and trusted code, this approach requires neither the program writer nor the compiler responsible for code in other compartments to be trusted.

Once correctly loaded, the code within domains defines their behaviour, so they can be effectively reasoned about. However, it is also necessary to verify that a given program (defined by some binary) has been correctly mapped into isolated domains at a system level. Domains can obtain a capability that contains a digest of the starting shape of their capability graphs when created, which can be used to attest to other domains that they were constructed as intended. See section 3.2 in this chapter for a brief description of this facility, and section 5.3 for details.

3.2 The nanokernel

The nanokernel could be seen as an ISA extension; it provides functionality that might be impractical to provide in hardware, but is otherwise relatively low-level. I propose that the nanokernel would be shipped in firmware alongside CHERI CPUs, and that kernels target CHERI+nanokernel processors (in the same way paravirtualised kernels already compose with hypervisors). This is comparable to the SPARC hypervisor[101], or Alpha's PALcode.[30] It is also similar to how microcoded processors are already designed, allowing hardware designers to implement whatever is most efficient, while handling more complex functionality at another layer. For example, the Cambridge CAP interprets all of its data capabilities in microcode.[96] Some Lisp machines implemented common functions in microcode.[14] Contemporary Intel machines save and restore context in microcode to facilitate enclaves.[32] A nanokernel resides at a similar level, and programmers need not distinguish between the nanokernel and hardware, which may change between hardware revisions.

The nanokernel serves three purposes:

Split Authority Provide a finer-grained set of architectural capabilities (such as those to manage virtual memory) than underlying hardware.

Reduce Authority Restrict the sum power of all architectural capabilities available to software in the system, so that, even holding every capability, some authority is withheld.

Provide Primitives Provide new functional primitives for improved security.

The first two of these allow creating least-privilege compartments with distinct authority. To perform any system operation, such as page-table manipulation, a nanokernel capability is required. This allows the nanokernel to enforce invariants beyond those of hardware. The last two allow us to reason about system-wide guarantees, even if the OS turns malicious. Nanokernel functionality offering isolation, integrity, and authenticity can be used directly by applications. See chapter 5 for details.

The nanokernel is significantly simpler than a microkernel in size, construction, and concept. At time of writing, the CHERI-MIPS nanokernel was 2625 handwritten MIPS *instructions*,

spread across ∼50 routines. It uses no call-stack, no scratch memory, has constant-sized state, and so makes no allocations. Every subroutine within is fixed-length and loop-free[1], apart from those that zero or sweep memory for revocation. Every nanokernel subroutine is preemptable, apart from one: a register-file swapping routine. Thus, even a real-time system could run atop a nanokernel. This contrasts a full microkernel, like seL4's, which comprises roughly 10 000 lines of C[2] and supports the full C abstract machine, in all its complexity.

The nanokernel does not try to provide a more usable hardware interface. While it does virtualise certain resources, a common or easily-programmable interface is not the aim; this is left to the microkernel. Where possible, every nanokernel routine maps directly to a MIPS architectural operation without further abstraction. All operations are allowed, provided they do not violate its security invariants. Increased stability is a non-goal; the microkernel is free to use the nanokernel to irrecoverably prevent the system from making progress. For virtualisation, another hypervisor would be required atop the nanokernel to further mediate access, so that no microkernel could dominate the system, although nanokernel primitives could be used to this end.

Primitives

A few abstractions are implemented by the nanokernel. Some extend naturally from hardware resources, and some are completely new. Capabilities are used ubiquitously to access and delegate access to these services. Software components freely share nanokernel capabilities. Some nanokernel capabilities are invocable, while others are bearer tokens, see section 2.3 for the distinction.

Reservations

Reservations (fully detailed in section 5.2) are central to CheriOS. Reservations allow software components to distrust others that delegate them memory, ensuring the delegator cannot also access the memory. They are used extensively throughout CheriOS to pass memory from the OS to applications and back (see figure 3.2 for an example). A reservation is the authority to allocate memory, not to access it. Reservations are 'taken', providing access to the memory they reserve. The resulting memory capability is guaranteed to be non-aliasing and unique system-wide. Uniqueness and non-aliasing of objects is enforced (by the nanokernel) even taking the MMU into consideration by ensuring invariants on pagetable management (see section 5.2). A reservation is taken atomically, and only once. Once taken, the memory capability gained is private, even if copies of the reservation exist elsewhere. The only way the memory can be shared is by delegation. Eventually, used reservations are collected, merged, and revoked by the OS.

[1]Not including loops that implement compare-and-swap, which sadly CHERI-MIPS lacks. Load-link / Store-conditional style instructions can to lead DoS attacks[35] as progress cannot be guaranteed. Contemporary architectures tend to feature richer atomics.

[2]Compiling to ∼38 000 RISC-V instructions or ∼40 000 AARCH64 instructions.

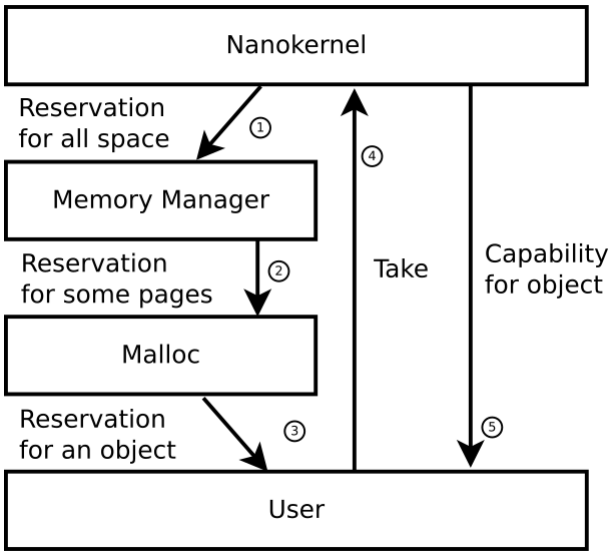


Figure 3.2: *Using reservations to allocate objects for the user*

Many capability OSs, such as seL4 and Barrelfish, also use capabilities to represent memory spans, and divide, delegate and then retype them in a fashion similar to CheriOS. CheriOS differs in both the granularity at which it uses them (as capabilities to individual language-level objects, not just pages and kernel objects), and moving much of the management (such as delegation and validation) of these capabilities into hardware. See section 5.5 for an in-depth comparison of CheriOS reservations with untyped capabilities in seL4.

Foundations

Foundations allow attestation of software, based on known starting states. Foundations were inspired by SGX enclaves, and solve similar problems, but have several crucial differences. An enclave is a hardware-enforced trusted execution environment, which protects a segment from being read or modified; see section 2.2.

Like an enclave, a foundation begins with a segment that cannot be accessed from the outside, and a cryptographic hash of the segment gives the foundation an identity. Creating a foundation mints an *entry capability*, which can be used to enter the segment at a pre-defined offset. The similarities between foundations and enclaves end there. Entering a foundation also grants a capability to act with the authority of the foundation. Acting as a foundation is not an ambient state, it is the property of having this *authorisation capability*. Authorisation capabilities can be arbitrarily delegated, so foundations are not confined to their initial segment. A foundation can release capabilities to its initial segment, so foundations are only necessarily enclave-like when created.

The other major feature offered by foundations is how authority capabilities are used. Authority capabilities can be used to sign other capabilities with the identity of the foundation, or unlock capabilities intended for receipt by a particular foundation identity. These primitives

are well understood for use in secure communication protocols. They can be used to treat the microkernel's IPC mechanism, as well as other IO channels, as untrusted communication channels, without having to resort to cryptography. Even more importantly, unlike if they were serialised and encrypted, capabilities protected this way maintain their architectural meaning. More can be found in section 5.3.

CPU contexts

The nanokernel provides an opaque handle to virtual *CPU Contexts*. Handles are sealed memory references saving a register file. CPU contexts facilitate switching without revealing CPU state. Allocating contexts, tracking them, and deciding which to save and restore is not managed by the nanokernel. Exceptions and switching are covered in more detail in section 5.1.

Physical & virtual pages

Unfettered management of memory is too powerful a capability to grant to the OS. Instead, state for every physical and virtual page in the system is kept and guarded by the nanokernel. Modification is only ever made at the behest of the OS; the nanokernel is interested in maintaining certain invariants, but will otherwise allow any operation. These invariants enforce the requirements of reservations, even with a misbehaving OS. Details are in section 5.2.

3.3 The microkernel

The CheriOS microkernel is only distinguished from another service or application in that it holds different nanokernel capabilities. Specifically, these capabilities deal with interrupts (interrupt masking, specifying an exception context) and CPU contexts (creation, switching, and destruction). In every other way it is unprivileged, running in the same ring as other components. It offers exception handling, time sharing, and fixed-length message-passing-based IPC to the rest of the system.

CheriOS bases its scheduling unit on the *object activation* proposed by Watson and Barral. Object activations are based on *scheduler activations*.[8] In the original work, scheduler activations are virtual processors offered by the kernel on which users build their own threads. The kernel explicitly notifies user schedulers when activations are de-scheduled, so new scheduling decisions can be made. This is an example of hybrid threading, where some M user threads are mapped to N kernel threads. Object activations add to this model by associating a multi-writer, single-reader message queue with each activation.

Several types of object activation references exist, each being a sealed CHERI capability pointing to kernel-managed objects. A *control reference* can terminate an object activation or enter the microkernel with the authority of that activation. *Standard references* are used to send messages to and get information about object activations. *Reply references* are single-use references to reply to synchronous messages. *Notification references* are used to resume activations sleeping on a notification. As these are all just capabilities in tagged memory, users can freely

pass them by value to each other. This model is similar to Mach ports, but binds a port to each scheduler entity. Binding message sending and scheduling simplifies the microkernel.

The focus of CheriOS is not scheduling, so only a primitive version is implemented. There is very little information flow between the microkernel scheduler and userspace, and userspace simply implements a 1:1 threading model. The current model is closer to the actor model than scheduler activations; most user threads handle the message queue of the activation they are bound to, selecting a function based on a selector in the message. Improving the userspace scheduler, and increasing information-flow from the microkernel scheduler, is future work. The primitive offered by the nanokernel (that of saving and restoring registers) has been designed with more complicated schemes in mind, so exploring different scheduling options should not require modifying the TCB.

An object activation can select any memory region it has a capability for to receive messages (although messages also arrive not via shared memory). It is able to dequeue messages without calling into the kernel, making message receipt efficient. A standard reference is used to send a message, either asynchronously, or synchronously. Synchronous messages mints a delegable reply-reference capability that authorises sending a response. Interrupts are also delivered as messages, so the same pattern is used to write a driver as any other service. A single message is sent when an IRQ is triggered (per IRQ source), and no more are sent until acknowledgement.

CheriOS encourages an event-driven, message-based approach to both OS and application design. Recent work on building high performance OSs for heterogeneous architectures, such as Barrelfish[113][104][13], emphasises this abstraction maps better to contemporary hardware than shared-memory models, as hardware behaves more like a message-passing network than a uniform memory. Barrelfish eschews the use of shared state, proposing a 'Shared nothing' model, as shared memory scales badly to larger systems, which may not be able to provide a single coherent and homogeneous address space. Although an asynchronous message-passing system at is core, CheriOS *also* makes heavy use of shared-memory objects for both flexibility and performance, using messages to exchange capabilities to objects, rather than sending objects themselves. Sharing a capability to a data structure with another process, relying on CHERI rather than the MMU for protection, is often more flexible and of higher-performance than other forms of coordination. A single address space is considered crucial in CheriOS's design, as hardware-defined capabilities must have a consistent meaning in any context, avoiding unexpected privilege amplification by delegation. Failing to do so requires programmers to reason about where capabilities come from, which contradicts the principle that a capability alone entails authority. Homogeneity of memory accessibility is not required in a single address space, only that, if accessible, CHERI capabilities must refer to the same memory regardless of context. The exact places where shared memory is used (i.e. for queues), are not a fundamental part of CheriOS's design.

3.4 The system

The microkernel performs no allocation and has no notion of memory ownership, threads, processes, or compartments. The burden of managing resources is pushed out to a set of services, or, where possible, is made the responsibility of user programs. Dividing out memory allocation is not a new idea, capability systems such as Eros[118] and seL4 both take the same approach. On CheriOS, applications can have memory with lifetimes unrelated to object activations. However, many applications expect a POSIX-like environment, that is, a process marrying some memory with some threads of execution. Some core CheriOS services provide this abstraction, which are communicated with by sending messages to particular activations:

Namespace manager Used for discovery of services, initially a new activation will only be able to contact this service.

Memory manager In charge of page-granularity memory allocation; see section 4.3.

Type manager Similar to the memory manager, but allocates CHERI sealing capabilities. It manages a type-space in the same way the memory manager manages a memory-space.

Process manager Creates the POSIX-like process abstraction. Its use is not mandatory, but most programs are loaded by it. It creates object activations for every POSIX thread, and mediates a session with the memory manager.

Event Dispatcher Used to subscribe to system events, such as activation termination.

Most programs run using the process manager, so a *CheriOS process* is a set of threads (implemented using activations) and a memory manager session, all managed by the process manager. Both the nanokernel and the microkernel are agnostic to the existence of these services, and other models could be built atop them. When 'CheriOS' is referred to, generally this means the entire software system: the nanokernel, microkernel, and system services.

Device drivers are also compartmentalised programs. They differ from others in that they hold capabilities to devices; see section 6.2. CheriOS has drivers for a few Ethernet, block, and UART devices. Abstractions on top of these devices, such as networking stacks and filesystems, are provided by additional compartmentalised services. These services are only special in that they receive capabilities to communicate with drivers directly. CheriOS uses a port of LWIP[45] for its networking stack, and of a simple FAT implementation for its filesystem.

3.5 Userspace

There is no userspace as such in CheriOS, but we might consider those programs that have relatively few nanokernel and microkernel capabilities, and no device capabilities, as 'userspace' programs. CheriOS supports automatic compartmentalisation on dynamic-library boundaries.

Userspace programs interact with both the nanokernel and microkernel by linking with them as dynamic libraries, and using standard calling conventions instead of system calls. These calling conventions not only protect the kernel from the user, but vice versa. These conventions are discussed in chapter 5. Other dynamic libraries, such as the IO-library, are also linked by most programs. The IO-library is used by practically every application for high-bandwidth communication; see Chapter 6.

The CheriOS API is different to that of UNIX, however, some abstraction layers are provided, allowing UNIX programs can run on CheriOS. These abstraction layers often sacrifice either flexibility or performance, but allow a security upgrade without programmer effort. For the NGINX case study in section 6.3, parts have been selectively re-written to take advantage of CheriOS interfaces for performance reasons.

Chapter 4

Memory Safety

CheriOS seeks to address several memory-related vulnerabilities through a combination of hardware and software techniques. CheriOS improves security by offering better guarantees when dereferencing memory in C than 'undefined behaviour' (such as making an out of bounds array access), by converting such accesses to fail-stop conditions. It builds on the previous CheriABI[37] work, which uses compiler and linker tools to implement C/C++ pointers as CHERI capabilities. Whereas CheriABI is primarily focused on userspace code with a trusted OS, CheriOS treats MMU and memory management as adversarial, in order to reduce the TCB. Without doing so, the entirety of the OS (including memory management subsystems) and the application memory allocator itself would have to be included in the TCB. CheriOS adds temporal safety on top of the spatial safety that CheriABI offers, including on the stack. This is achieved by extending the existing CHERI compiler/linker work and introducing new mechanisms at both a system and user-library level.

Denial-of-service, or attacks using timing or fail-stops as side channels, are out of scope. In section 4.4 I evaluate the performance implications of any new mechanisms introduced, and compare to related work (section 4.5). The focus of this chapter will be safety in C; however, the primitives are all implemented at a machine-code level, so will affect other compiled languages.

4.1 Memory allocation

Objects abstract raw memory, associating a region of data storage[65] with a single entity, called an object. Objects have a lifetime, so bytes are reused for different objects. At different times, the same address does not necessarily correspond to the same object. Memory allocators provide a reference, on demand, to a memory region when a programmer requests an object. An OS will have different allocators present at different levels of abstraction. For example, an OS may have a page allocator responsible for providing page-aligned memory spans. Individual process-level allocators request memory from system-level allocators, and allocate objects corresponding to source-language objects. A common interface, that used in C, is to specify how many bytes are required (as opposed to a type, for example) when allocating an object. The lifetime begins at the

point of allocation, and the end is specified explicitly using a `free` function call (or destruction of the containing process).

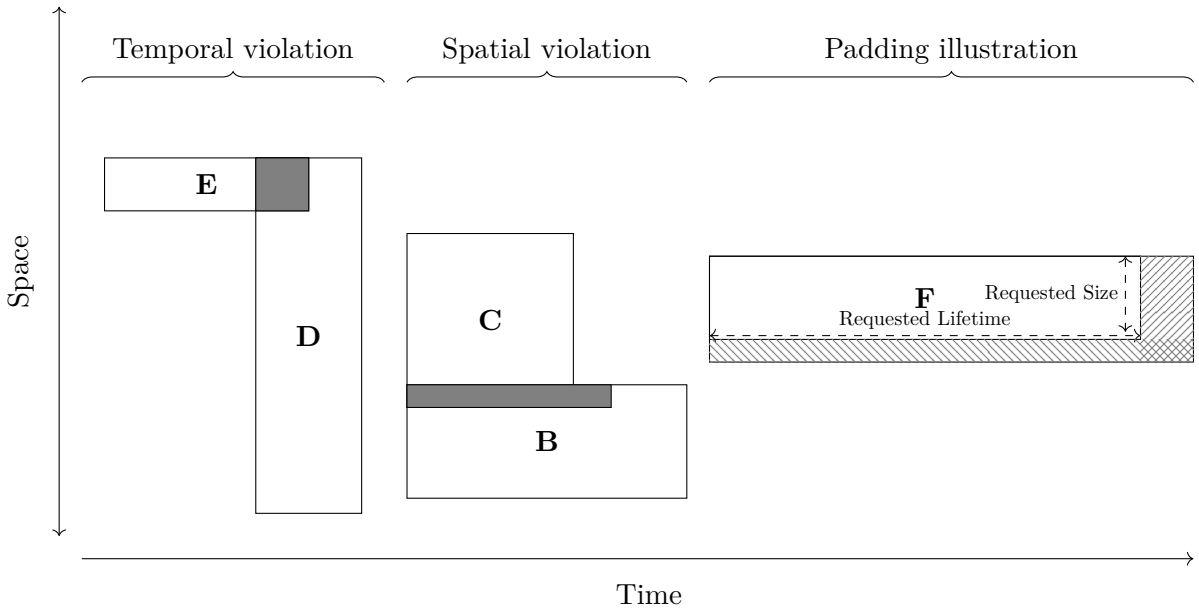


Figure 4.1: *Figure shows a memory over space and time. Objects, boundaries we enforce on the references to them, and potential violations are also shown. A reference to **C** accessing **B** is a spatial violation. A reference to **E** accessing **D** is a temporal violation. Object **F** has additionally been labelled with the exact size and lifetime a user requested. Additional space is padding.*

In CheriOS, we desire system-wide memory safety. There are two forms of memory safety: spatial and temporal. In both cases, violating safety means a reference meant for one object is used for another; this confusion causes error. A reference should never be able to access objects other than a single intended one. Figure 4.1 depicts the two kinds of possible violation. A buffer overrun (i.e. when a pointer is incremented past the end of an array into the next object and then dereferenced) is a violation of spatial safety. A use-after-free (i.e. calling `free` on a pointer and then dereferencing it after the allocator has re-purposed the memory) is a temporal violation. This model takes a simplistic view of memory, which is not accurate from a system perspective.

Although convenient to think memory one large linear space with different locations referring to different stores, in reality, addresses undergo translation by the MMU. MMUs allow aliasing to occur both spatially and temporally. Aliasing is where two apparently distinct references refer to the same object. MMUs can cause two different virtual addresses to alias in space, even if they appear not to, by mapping two virtual pages to the same physical page. Alternatively, MMUs can cause the same virtual address to not alias with itself, at different times, by changing a mapping. CheriOS extends temporal and spatial safety to include MMU behaviour, protecting against the same class of vulnerability, but when the attacker has MMU control.

On CheriOS, two different virtual addresses aliasing spatially (at the same time) is a spatial violation. Failing to have a virtual address alias with itself for the lifetime of an object is considered a temporal violation. Furthermore, CheriOS considers aliasing the same virtual

address at different times, but across the lifetimes of *different* objects, a temporal violation. In other words, changing a mapping during an object's lifetime, or failing to map to zeroed memory at reallocation, are temporal violations in CheriOS. Together, these rules ensure references only allow access to a single object, which aliases with no other, and cannot change unless modified via that reference.

Note that neither the lifetime, nor size, of an object an allocator provides may exactly match what was requested. CheriOS does not consider this a memory-safety violation, as long as one object reference still cannot be used to access another. The most common case is the insertion of padding. It is often easier for allocators to round up sizes and extend the lifetimes of objects, as long as this does not cause intersection with another. Padding is considered part of the object from the perspective of an allocator. Objects may also be smaller than anticipated by a programmer. For example, the OS may prematurely end an allocation's lifetime in response to critical memory pressure, or a process being apparently non-responsive. Applications are expected to handle the resulting exceptions, which can be handled on a compartment granularity; see section 5.1.

CheriOS has a single virtual address space. Furthermore, processes freely share objects from their individual allocators, so memory safety must be enforced system-wide to account for untrustworthy allocators. CheriOS provides its enforcement dynamically using CHERI. Although it imposes restrictions on its usage, it is unreliant on the MMU for protection. Every application has uniform access to virtual-page mappings. This is in contrast to previous OSs, where dynamic enforcement is achieved using the MMU.

C pointers, which are returned by allocators, are implemented using CHERI capabilities. Capabilities are mandatory for memory access. Allocators need to bound capabilities they provide in both space and time. CHERI provides capabilities with explicit spatial bounds, so it is easy to enforce spatial bounding in the allocator via their use. Temporal bounding in the past direction is also trivial. Capabilities that have provenance of an allocation cannot be used before the time of allocation, as capabilities cannot be derived from one that does not yet exist.[1] Future temporal bounding is harder, as object lifetime is not known at time of allocation. Future bounding is achieved via revocation. Revocation is discussed in section 4.2. CheriOS ensures the rules for MMU aliasing are followed using *reservations*.

Reservations, a nanokernel primitive introduced in section 3.2, facilitate memory-safe allocation. Reservations can be passed to a nanokernel *take* routine, which returns a memory capability. The result of taking a reservation is guaranteed to provide a memory capability with bounds that cover no other allocation, and to properly handle all the rules for aliasing, in both space and time. Only the nanokernel and CHERI must be trusted; no other software component is required to cooperate in order to guarantee these properties. If a single reservation is used directly for a single object, the result is already guaranteed to be both spatially and temporally safe. Reservations, and how properties are enforced, are covered in more detail in section 5.2.

[1] Time machines are considered out of scope.

In C, objects are allocated in one of three places: the heap (`malloc`), stack (local variables or `alloca`), or static storage (functions, global and thread-local variables, and local-static variables). Each is handled differently, but strategies for ensuring memory safety are similar in each case.

There is no requirement that a programmer use the CheriOS built-in safe allocators. In fact, I observed a common practice when porting software to CheriOS that C programmers often write custom allocators atop malloc. Such programs may very well still exhibit memory-safety vulnerabilities, but they cannot induce them in others, even if they share allocations. Conversely, programmers can verify using attestation (covered in section 5.3) that intended libraries are in use, and no extra defects were inserted during load-time. Compilers can be run on a more trusted system than the system used for deployment, so even though we require its correctness, a minimal TCB is maintained on CheriOS. In the common case where the default allocator is used, programs gain the memory-safety of running on CheriOS with no programmer effort.

Static storage

A common linkage model has the linker place offsets to global variables in a table, known as the GOT (global offset table). Programs access globals by GOT index, rather than absolute address, which is useful when their runtime location is unknown. The previous CheriABI work[37] offered spatial safety for global-variable and function storage using *captables*. A CheriABI captable is an array of capabilities, one for each object, similar to a GOT from a conventional runtime, but with bounded capabilities rather than integer offsets. The memory for global storage is mapped as one block by the OS, and a capability to it is provided to the dynamic linker. When loading a program, an instance of the dynamic linker populates the captable by processing the relocations specified in the program binary. When running, accesses to globals first load a capability from the captable, and then dereference it. This ensures a spatially bounded access is always made. It should be noted that the captable is only ever indexed via a static value. It is therefore not subject to adversarial data injection.

CheriOS extends this scheme to provide stronger memory-safety guarantees and reduce trust. Firstly, the block of memory used for global storage is given to the program via a reservation. This ensures not only temporal safety, but removes the need to trust MMU management outside the nanokernel, and ensures the OS cannot access the memory. CheriOS does not trust a dynamic linker to process relocations for the captable, or trust that relocations have not been tampered with before loading. Binary loading is checked using the attestation techniques covered in section 5.3. Because software other than a program itself should be unable to access capabilities in a captable, relocations are handled at runtime. This is done by statically embedding a small part of dynamic-linking logic in every program. This code is relatively simple, and cannot be modified without violating the checks provided by attestation. How exactly dynamic linking is handled is covered in section 5.1. After it has been subset, the capability for all static storage memory is discarded, and is not available to the main program. All accesses to functions and globals must therefore go through a captable.

CheriOS, unlike CheriABI, also uses a captable for thread-local storage. This was implemented via modification to the compiler, linker, and runtime libraries. The local captable is allocated and populated in the same way as the global captable, but individually for every thread. This ensures bounds are per-variable, not per-segment, for thread-local variables, and also provides some isolation between threads by default.

It is worth mentioning that captables defeat more attacks than buffer overruns, and CheriOS inherits these benefits. ROP attacks can no longer function just by overwriting the return pointer with plain characters (as would occur with, for example, a buffer overrun with string data). A gadget that could construct capabilities and overwrite the return address with them would be required. See the CheriABI work[37] for more.

Heap memory

In CheriOS, truly freeing objects is a multi-phase asynchronous process and is not handled by applications. Calling malloc's `free` function informs malloc that an object should no longer be needed, but will not necessarily make it inaccessible. Malloc, and the OS, will free the object at some future point. Attempting to access an object guarantees an exception only if freed by the OS; before then, access will succeed. However, as the object has not yet been truly freed, CheriOS will guarantee its contents will not be corrupted or aliased. It may seem counter-intuitive that calling `free` on an object and then accessing it is not considered use-after-free. The heap in CheriOS should be viewed as garbage collected, where the `free` call allows the collector to consider the object for collection.

This 'asynchronous free' view of temporal safety prevents memory-corruption bugs, but is not guaranteed to catch other logical errors. Stronger enforcement would be possible with CHERI, but would incur further performance costs. Stopping memory corruption is not only easier, but also of most importance.

The CheriOS C-runtime allocator uses a reservation for every allocated object, ensuring both temporal and spatial safety. In fact, the allocator returns a reservation, not a memory capability. A C-header wrapper converts between the two for a standard `malloc` call, see figure 3.2 for a diagram. Reservations are usable only once, which is enforced by the nanokernel. This means that even after freeing an object, malloc cannot reuse the same memory. Instead, malloc returns the memory to CheriOS's memory manager. The memory manager is designed to allow efficient continuous allocation of new memory as malloc requires it, only imposing limits on total in-use memory. The OS will eventually reclaim memory, and does so using a process called revocation. Revocation is covered in section 4.2.

Because the heap allocator returns a reservation, programs need not trust the allocator. Malloc never gets a capability to the objects it manages. The `free` call takes an address, not a capability, so malloc cannot even access program objects accidentally. This prevents exploits that rely on corrupting in-band heap metadata,[48] e.g. use-after-free or double-free.

CheriOS's current malloc implementation is relatively simple and unoptimised, using a slab allocator for small objects, and returning pages straight from the memory manager for large ones. Heap fragmentation is currently unsolved on CheriOS, and this is discussed further in section 4.2. Future work on CheriOS could include some of the fine-grained revocation techniques, as proposed in CHERIvoke[138] and Cornucopia[49], which are discussed in section 4.5.

Stack memory

Stack allocations and deallocations are made implicitly by compiler-generated code that runs on function call and return. Allocations are also made explicitly by calling the `alloca` function, but are still freed on function return. The stack itself is normally allocated on the heap, or via an OS-provided memory mapping.

As with the other allocators, providing spatial safety is done via setting bounds on CHERI capabilities. Performant temporal safety for stack memory is harder to provide than other allocation types, as stack allocations are made more frequently, and paying the same cost as heap allocations is untenable. Typically, stack allocations and deallocations are made in a single cycle, and several are grouped into a single stack frame. `Malloc` calls tend to take dozens to thousands of cycles, and the resulting heap structures can become fragmented over time. In contrast, the free-order of stack allocations is restricted, and allocation usage can often be reasoned about statically, giving room for optimisation. CheriOS uses a novel stack structure, called 'Slinky Stacks', that facilitate temporally safe stack allocations. CheriOS also uses novel escape analysis to reduce the cost of slinky stacks even further.

As compartments in CheriOS do not trust each other, each compartment manages its own slinky stack. Transitioning between different stacks is handled by the ABI; see section 5.1.

Slinky stacks

Conventional stacks occupy a single contiguous memory region. Stack frames (a group of objects used for a function) are allocated and deallocated by subtracting or adding the size of a frame to a 'stack pointer'. *Segmented stacks*, conversely, occupy several discontinuous segments arranged as a linked list, see figure 4.2. Each segment is treated like a normal stack, subtracting or adding to a pointer to allocate or deallocate. The linked list is traversed to obtain a different segment if the in-use one is running out of space. Segmented stacks are normally used to provide variably-sized stacks when memory pressure is high. They were used, for example, by the Go compiler prior to version 1.3.[105]

We introduce a new structure, *Slinky Stacks*, a variant on segmented stacks that provide temporal safety via selective non-reuse. Slinky stacks still require very few instructions to manipulate and have far better bounds on fragmentation than a general heap allocator. The only fragmentation on a slinky stack is between segments, whereas heap allocators can be fragmented between each individual object. Furthermore, if static analysis shows a function makes no potentially temporally unsafe allocations, slinky stacks degrade into normal stacks, so incur no extra overhead beyond an unused register.

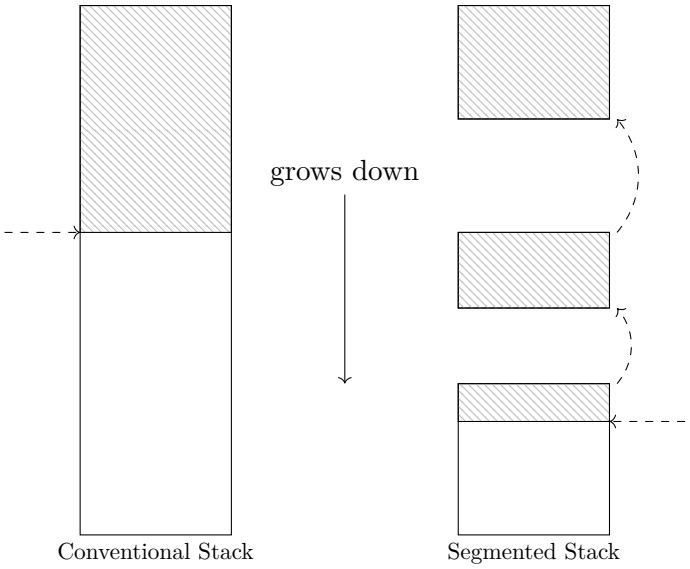


Figure 4.2: *A conventional stack vs. a segmented stack. Shaded sections show allocated space and arrows to corresponding stack pointers.*

Slinky stacks differentiate between allocations proven by the compiler to be used only in temporally safe ways, and those otherwise. All allocations are made on the same stack, but space is only immediately reused when usage has been proven safe. Space that might be used in an unsafe way is not reused, eventually being given back to the OS for revocation. Dangling references either continue to point to the same non-aliasing allocations (maintaining sensible contents and prohibiting corruption), or will trap upon use.

Types of allocation When allocating on a slinky stack, we differentiate between 'safe' and 'unsafe' allocations. *Safe allocations* are those where, assuming the correctness of the compiler, it is impossible for a reference to the stack object to escape beyond the lifetime of the function. No assumptions about the behaviour of other compartments is ever made. *Unsafe allocations* are those where safety cannot be proven statically. Over its lifetime, we desire each memory location hosts a succession of safe objects, then a single unsafe object, and then is never used again. This gives temporal safety, but still maximises reuse where safe. It is always sound to consider every allocation as being unsafe, but this would impact performance. It is only possible for an allocation to be unsafe if it is reference taken, so commonly allocations are trivially safe. In section 4.4, we report 88.31% of functions in NGINX make no unsafe allocations. The uses of a reference that make the allocation unsafe are: returning it from the function, writing it to a global, passing it to another compartment, or writing it to any location that is itself unsafe. Once outside of its lifetime, an object is considered *dead*, otherwise *live*. Capabilities to dead unsafe allocations might still be erroneously dereferenced.

Stack allocation can be further sub-categorised as either static or dynamic. *Static* means that the allocation is of a compile-time-known size, and need be made only once during a function call. *Dynamic* means that the size is unknown at compile time, or the allocation is performed an

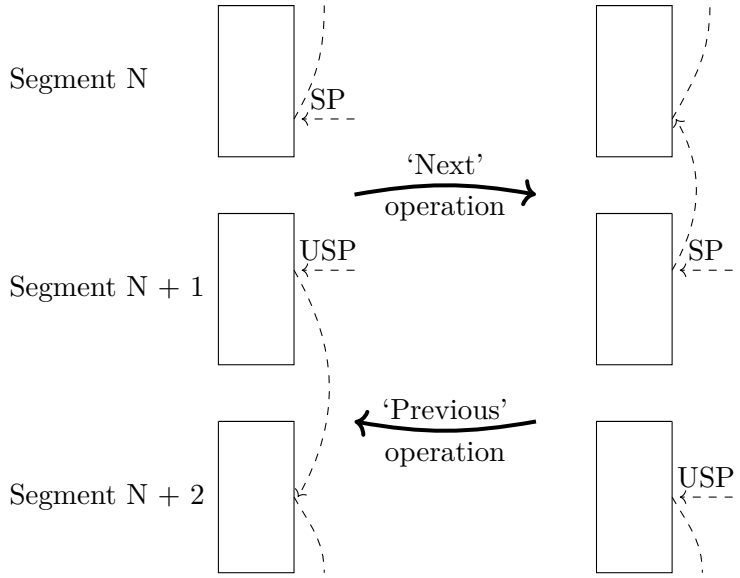


Figure 4.3: *The 'next' and 'previous' operations for a slinky stack.*

unknown number of times. In practice, dynamic stack allocations are very rare; many programs, and every program in the test corpus, never make any. Static allocations are common, they are needed on practically every function entry. We refer to functions as *unsafe* if they make any static unsafe allocations, and *dynamic unsafe* if they make any dynamic unsafe allocations.

Stack structure A conventional segmented stack allocates from only a single segment at a time. Slinky stacks differ in that there are two active segments in use by any given function execution. One is called the *safe segment* and the other the *unsafe segment*, where the function allocates safe and unsafe objects respectively. Rather than a single stack pointer, SP, we how have two: SP (the safe-stack pointer) and USP (the unsafe-stack pointer), which point into their corresponding segments. Both of these registers are 'live-in', meaning they are set up by the caller for use by the callee.

Safe objects are allocated by decrementing SP, and deallocated again by incrementing it by the same amount, allowing immediate safe reuse. Unsafe objects are allocated by decrementing USP, but are deallocated by *doing nothing*. This ensures that, if further objects are allocated, they will not occupy space used by a previously allocated unsafe object. An invariant of slinky stacks is that once space has been used for an unsafe object, it cannot be used again for another object. This ensures that any out-of-lifetime accesses do not access another object. Dead space can be unmapped (at a page granularity), immediately making physical memory collectable, which the OS does in batches across all processes.

A function's safe and unsafe segments are two adjacent elements of one linked list; see figure 4.3. Links are stored intrusively in segments themselves. The safe segment comes directly before the unsafe segment in the list. We now define two operations: next and previous. A *next* operation will shift along the linked list in a forward direction. That is, SP takes on the value of

USP, and USP will point to the next segment in the list. A *previous* stack operation does the opposite, USP is set to SP, and SP is set to point to the previous segment in the list. Both these operations are very cheap, requiring only 3 instructions: one move, one load, and one store. If there is no next segment, a trap is taken and one is created.

Not deallocating objects on an unsafe segment could cause fragmentation. To prevent this occurring, functions use next and previous operations to ensure only one frame's worth of unsafe objects are on a segment at a time. This makes dead regions compact and contiguous, so they can be unmapped.

Functions preserve SP on return (as they decrement and increment by the same amount), so its value is unchanged on function entry and exit. However, a function may modify USP on return, either pointing further along the segment (skipping space not safe for reuse), or pointing to a completely new segment. The invariants for slinky stack segments are:

1. Unsafe objects on a segment must not be allocated after live safe objects.
2. On entry and exit from a function, the unsafe segment and all segments after it must contain no live objects, although dangling references to them may exist.

Note that one function's safe segment may be another's unsafe segment, so all segments may contain both safe and unsafe objects.

Figure 4.4 depicts an example section of a slinky stack during two function calls, spread across three segments. The segments are shaded corresponding to the type of allocation present. Note that each segment starts with some dead space from previous usages of the segment, then some number (potentially none) of unsafe allocations, then only safe allocations, and then available space for more allocations. Notice a new segment is started only when unsafe allocations need to follow safe allocations, and when a function returns, unsafe space becomes dead space.

Static allocations On function entry, we first need to allocate all static objects. First we allocate a block of memory for unsafe objects on the unsafe segment by decrementing USP. This is allowed without invalidating invariant 1, as invariant 2 requires that on entry to a function USP point to an empty segment. If the function is unsafe, that is it had any unsafe objects to allocate, we then perform a 'next' operation to promote the function's unsafe segment to a safe segment. We then allocate one block on the safe segment for the safe objects, this time by decrementing SP. Note that, whether or not there were any unsafe objects, both blocks end up contiguously on the same segment because of the promotion. See in figure 4.4 the call to the unsafe function. SP points past both a block of safe and unsafe allocations. In practice, allocations are not made in two parts, both are allocated simultaneously. However, it is helpful to think of there being two allocations on two different segments, with a promotion in-between, so we will continue describing them in this way. The 'next' operation can be folded with the decrement, so the combined cost for allocating static objects is, at most, three instructions. All static objects, whether safe or not,

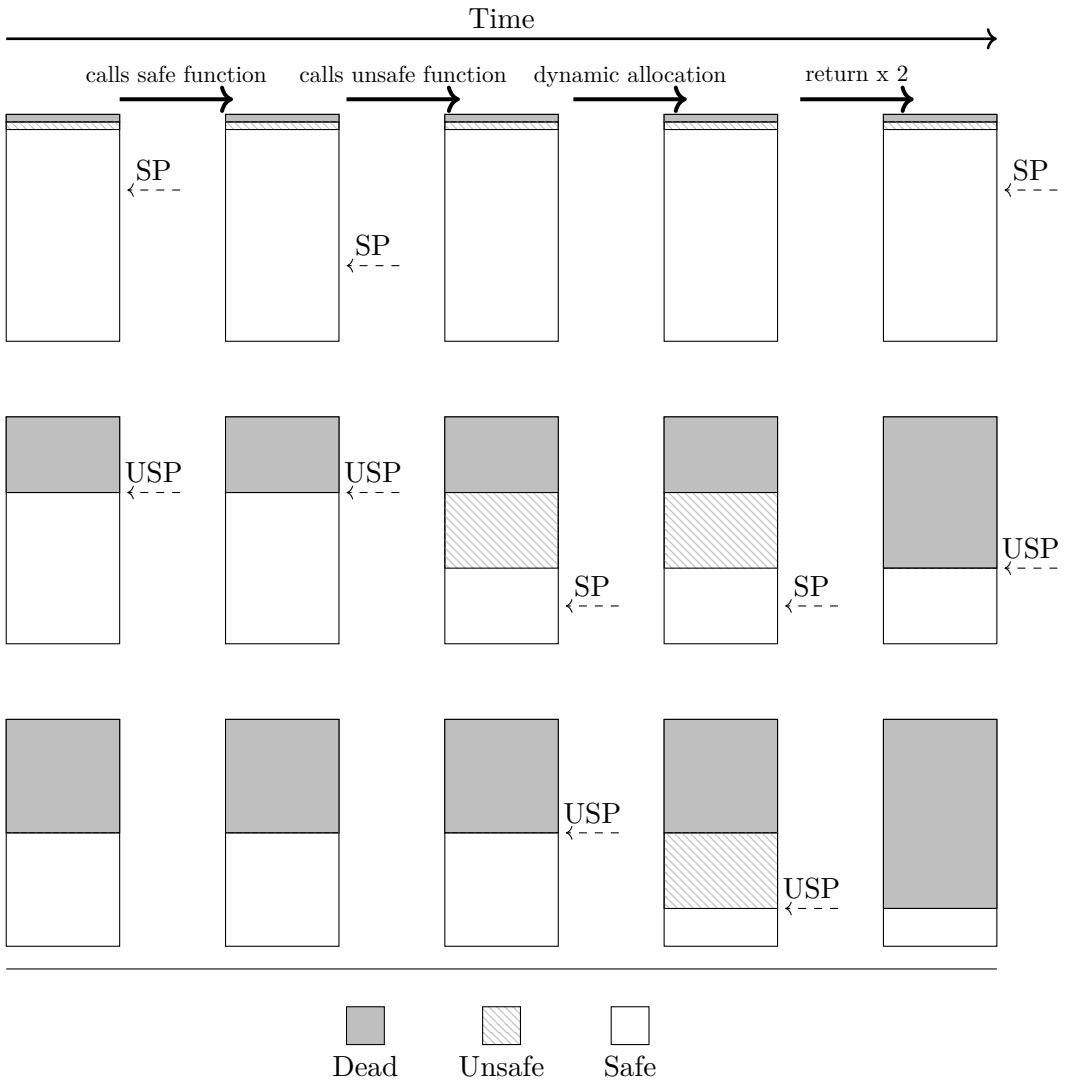


Figure 4.4: *A section of a slinky stack. Unsafe calls require a 'next'. Unsafe space eventually becomes dead. The first call is to a safe function. The second to an unsafe function that then makes a dynamic-unsafe allocation. The last stack state is after both functions return.*

are accessed via SP. After the function prologue, USP always points to an empty segment, either because no unsafe allocations were made, or because of the promotion.

Deallocation is performed differently than with a conventional stack. On function return, static objects are deallocated in inverse order. Safe objects are deallocated conventionally, by incrementing SP by their total size. If the function was unsafe, a 'previous' operation is performed, matching the earlier 'next'. Then all static unsafe objects are deallocated; this is a no-op. USP is left pointing just after the now dead unsafe objects, as to ensure temporal safety that space should not be reused. The 'previous' operation is folded with the SP decrement, so the cost of all static deallocation is three instructions. As USP will have no live objects on it, the function can return. USP may have been modified by the function call, pointing it slightly further along

the stack segment than it did on entry. Subsequent calls to other unsafe functions will therefore not re-use segment regions that ever contained unsafe objects.

Dynamic allocations Dynamic safe objects are simple to allocate: they are put on the safe segment by decrementing SP. It is always allowed to place safe objects after any other type. To deallocate them, SP is restored to its initial value. Dynamic unsafe objects are allocated by incrementing USP, which will not violate invariant 1 as the unsafe segment should only contain unsafe objects. Deallocation is still a no-op. See again figure 4.4 and look at the dynamic allocation. After a dynamic unsafe allocation, the unsafe segment will not be empty, so invariant 2 would be violated if another function were called. Instead, a second 'next' operation is performed before any calls, either in the prologue (as we did with static functions), or before every call.

It should be noted that programs that never make unsafe allocations will put every stack object on one segment, and never perform a 'next' or 'previous' operation. Because of this, slinky stacks degrade into normal stacks when usage is proven safe, apart from an unused register. In the worst case, a function requires two 'next' operations. One for static unsafe objects, one for dynamic unsafe objects. How many segments a slinky stack needs therefore grows with maximum *unsafe call depth*, the number of call frames on the stack that require unsafe objects. This cannot grow boundlessly, else even a normal stack would overflow. Thus, maximum fragmentation is limited to, at most, the largest call depth a program can achieve, doubled if dynamic allocation is used. In practice, there are far fewer fragments, and even fewer in active use; see section 4.4.

As memory for unsafe objects is not re-used, the live part of segments migrate through memory. A segment's rear end moves downwards through memory when their front end moves upwards, giving rise to the name 'slinky stack'. Crucially, this leaves contiguous dead objects. When performing a 'next' operation, a conditional trap is taken if the segment is running out of space (or is non-existent). If this trap fires, the offending segment will be empty (by invariant 2), containing only dead objects, so can be unmapped. No objects need moving as there will be none to move. Because functions can return with a different USP, callers are agnostic to whether stack segments are being consumed.

There is some memory wastage when using slinky stacks. We expect, on average, half of every page and cache-line at the boundary between stack segments to be unused. This wastes not only real physical memory, but cache-line and TLB entries which leads to reduced performance. Furthermore, the individual segments are each large enough to hold an entire stack plus a good number of dead objects. However, not all of this memory needs to be backed at once. In practice, most segments are small. Also, even though each segment is larger than a conventional stack, only a stack's worth need ever be mapped. Memory that is not accessed costs no TLB or cache entries. On a low-memory system, we might consider reclaiming dead pages on a finer granularity than segment replacement. As a segment migrates through memory, we could trap as new pages are first accessed. Given the stack should only be of a given maximum size, at the point we need a new page, a dead one should always be unmappable. The current implementation forgoes this optimisation. Physical memory is committed for the entire stack segment and then reclaimed

all at once upon depletion. Currently these stacks are ∼1MB in size. The number of segments, which will dictate wastage, does not grow all that large in practice; see section 4.4.

CHERI-aware escape analysis

The CheriOS compiler automatically decides whether allocations are safe or not, although new 'safe' and 'unsafe' keywords were added to allow programmers to override its behaviour. Use of the safe keyword is potentially dangerous, as it overrides the automated pass. Its use is limited to actual unsafe practice (such as passing a reference across a compartment boundary), where a high-level invariant gives the programmer knowledge doing so is safe. For example, arguments to the nanokernel are marked as safe. The unsafe keyword is only useful if, for whatever reason, the automated pass is disabled. LLVM already contains a capture-tracking analysis pass, which is sufficient for declaring most stack allocations safe. However, because the CHERI architecture imposes restrictions on how capabilities (and thus pointers) can be derived, it is possible to do better. CheriOS implements a custom analysis pass which statically verifies more allocations as safe than the default pass, and the two are used in conjunction. Implementation details of the pass are in appendix C, and its efficacy is evaluated in section 4.4.

4.2 Revocation

CheriOS achieves temporal safety by virtual address non-reuse. Sadly, this cannot be done indefinitely. Addresses are fixed size, so there are finitely many nameable objects. Eventually, addresses must be reused, but only when the nanokernel can guarantee that no more capabilities are constructible with provenance of the original allocation. On non-CHERI systems, this is fundamentally impossible, as pointers to memory are arbitrarily constructable. Non-CHERI programs can reconstitute pointers hidden in data, or even from information stored outside the machine. However, on a CHERI machine, hardware enforces rules on capability derivation. As CheriOS pointers are capabilities, this in turn limits how a program can construct pointers. If the nanokernel removes every copy of a capability from any place it may be stored (apart from private nanokernel locations), then the system cannot derive a capability to memory with provenance of the old allocation. Programs can still keep non-capability data, such as addresses, but these are not usable for accessing memory, so temporal safety is assured. Logical errors concerning address reuse, for purposes other than dereferencing, remain possible.

Removing all instances of a capability from a system is potentially expensive. With CHERI, capabilities are stored not just in registers but anywhere in memory. Revocation requires searching all of memory and, although unlikely, could require modifying every memory location. Instead of performing this expensive revocation whenever physical memory is depleted, we instead use the MMU to revoke access to only the *physical addresses* by way of changing pagetables, while still allowing capabilities to virtual addresses (that will fault on dereference) to remain. We need use the expensive full-memory sweep only if virtual memory runs out. Virtual memory is much larger than physical, so we wait longer between revocation passes.

Combining these techniques allows us to perform frequent, fast MMU-revocation, and infrequent, slow sweeping-revocation operations. Both are performed by the nanokernel, at the behest of the OS. Users are assured that revocation is utilised properly due to reservations, see section 5.2. How the nanokernel tracks the state of both virtual and physical pages can be found in section 5.2.

MMU-based revocation

The first and faster technique used to revoke access to physical memory is un-mapping virtual pages. Virtual memory is used by everything but a small set of OS compartments that cannot tolerate virtual memory traps, due to an implementation quirk.

When the OS finishes with a page, it informs the nanokernel, which marks the physical page 'dirty' and the virtual page 'used' in corresponding tables. The TLB may contain stale entries, and the MMU-based revocation is only considered finished when a later shootdown of the TLB has been performed. This asynchronicity is safe with respect to CheriOS's view of temporal safety; use-after-free is allowed as long as an object does not alias. Shootdowns are batched, performed only when dirty physical pages are cleaned; until then, stale entries in the TLB can keep using dirty pages as contents will still be valid. For simplicity, where shootdowns are batched, the nanokernel clears the entire TLB rather than track revocation.

Physical pages, once cleaned, are reused immediately. This is fundamental in delaying sweep revocation. Virtual pages are unavailable for remapping until they are recovered by a sweep. Capabilities are allowed to exist that have MMU-revoked backing, but they will fault on dereference. They are still tagged, and valid bearer tokens.

Requiring page-granularity revocation causes fragmentation. At present in CheriOS, we just accept fragmentation. This is somewhat combated by making special effort to keep dead objects contiguous, for example on the stack, see section 4.1. Only the heap poses a problem. Programs written specially for CheriOS, such as drivers, are well aware of fragmentation, and are designed to bound it. Long-lived user programs that call `malloc` with objects of wildly different lifetimes would cause problems. There exists research on both reducing overhead of non-reuse allocators,[3] and performing small-granule revocation on CheriBSD.[138] Applicability of the CheriBSD work to CheriOS is discussed in section.4.5

Sweep-based revocation

Eventually, even virtual memory runs out. This will happen relatively infrequently: CHERI capabilities can hold 64-bit addresses, and CheriOS currently uses a 40-bit (1TB) virtual address space.[2] Upon nearing exhaustion, a virtual range is recovered by sweeping all of physical memory to revoke any surviving capabilities.

[2]MIPS uses the top 2 bits of an address as a selector, so a 62-bit space (4EB) is possible, although the current CHERI-MIPS implementation supports only the 40 used by CheriOS. The number of bits available on other architectures varies: contemporary x86-64 implementations by either AMD[40] or Intel[58] tend to support 48-bits (256TB).

The naive approach to remove all copies of a capability from a system would be for the nanokernel to loop through every memory location (or those in use) and atomically untag[3] any capabilities it finds that are subsets of the revoked range. The issue with the naive sweep is that already-swept memory might be concurrently re-dirtied.

Revoking capabilities shares some similarities with garbage collection. It is easier as applications calling `free` declare which objects need collecting (so no marking is needed). However, we also suffer some very particular constraints. Generally, a garbage collector will be concerned with collecting objects from a single program, which has a limited memory footprint. Furthermore, the program is cooperative with the collector. The case in CheriOS is quite different. Not only are we trying to remove references system-wide, but programs running on CheriOS are mutually distrusting, and quite possibly malicious, and so may try to hide capabilities from a naive sweep.

'Stopping the world' was ruled out as an option due to performance implications. If it were just one process being stopped, a brief pause might be fine, as systems regularly pause processes for time sharing. Stopping the entire system to perform a sweep on CheriOS could be prohibitive, especially as the space to sweep is for all applications, not just one.

Instead, I extend the CHERI ISA to offer hardware-assisted revocation. One advantage leverageable over traditional garbage collection is that, using the MMU-based technique to delay revocation, the objects that need revoking will have already been compacted. As we do not care about which ranges we recover, we can always target the largest ones, which are probabilistically large, considering the ratio of virtual space to physical space. Because there will not be many ranges to revoke, information on what needs revoking can be cached right next to the CPU pipeline.

I introduce a new triple of CSRs. A *filter triple* is made up of three integers: a bottom, a top, and a permissions mask. On the store path of the processor, if a capability is stored that has a base where $bottom <= base < top$, and has permissions that are a subset of the filter permissions, the stored capability is untagged by hardware. The permission subset is required to not conflate memory capabilities with other sorts, such as sealing capabilities. Revoking memory locations 1000 to 2000 should not result in sealing types 1000 to 2000 also being revoked! To revoke memory, all memory-related permissions are put in the permission filter. To revoke a sealing type, the seal, unseal, and pseudo 'sealed with' permission bits are put in the permissions register. The 'sealed with' permission bit specifies that any capability sealed with a type where $bottom <= type < top$ should also be revoked[4].

[3]There are actually several options of how to handle capabilities that needs revoking. Programs that rely on the value of freed pointers are incorrect, according to the C standard[65] such values are indeterminate, but it is best to provide minimal disruption. Zeroing is a poor solution as this loses all information for debugging. CheriOS chooses to untag, as this removes all privilege, but maintains all other information. The CHERIvoke work for CheriBSD suggests unsealing the capability and clearing permissions. This still removes all privilege, but maintains the tag at the cost of the type and permissions. Finally, we might introduce an explicit 'was revoked' bit, if we really wished to preserve all information.

[4]The benchmarks in this dissertation fully implement the sweep, but the filter registers only exist in simulation. Setting them up is a trivial cost compared to the sweep. CPUs already contain a number of 64-bit registers, and so an extra three should not be prohibitive. Whether the 64-bit comparison would affect the critical path of a given processor design has not been explored.

The comparison has similar complexity to existing 64-bit comparisons performed by a processor, and can be performed anywhere between register-file read and memory write-back. On a RISC architecture, comparable to CHERI-MIPS, this leaves ample time. The mechanism is also exception free, and will not change whether or not a store succeeds. This is desired behaviour as copying dead capabilities around should not be illegal as long as they are never used. These filter triples are accessed like other CSRs, and only by the nanokernel.

In order to perform a revocation sweep, it suffices to set up the filter registers on all cores for the capabilities that need revoking, and then perform the naive sweep[5]. Any capabilities written 'behind' the sweep will be untagged in hardware. In more detail, the steps for sweeping revocation are as follows:

1. Set filter registers on all cores for the virtual range to be revoked.
2. Set any PTEs for the virtual range from 'Unmapped' to 'Freed'. Any pages still mapped in the range trigger an error.
3. Broadcast an IPI to all cores to shootdown the virtual range from their TLBs, and also flush their L1 caches. TLB shootdown is necessary as MMU-based revocation may leave stale TLB entries.
4. Perform a synchronisation barrier on the TLB shootdown, flushing, and filter registers being set.
5. Scan over in-use physical memory, loading only tags, to see if there are capabilities. If one is found:
    5.1. Load the capability. Check if it needs revoking, if it does:
        5.1.1. Perform a compare and swap, with both old and new being the loaded capability. If the CAS fails, then the location has changed, and filter registers will guarantee it no longer needs revoking. If CAS succeeds, filter registers will de-tag the value being stored. Thus we can ignore the result of the CAS.
6. Cause a context switch on every core, this saves and restores every register to memory, and so revokes any capabilities in register files.
7. Clear filter registers on all cores. Perform a synchronisation barrier to ensure all cores have done so.
8. Set entries in pagetables for virtual range to 'Unmapped'.
9. Create new reservation for the range and return it.

On a single-core system, the broadcast and barriers can be skipped. Note how we try to, as much as is possible, avoid using atomic operations. Even without a write-through L1, flushing

[5]Like most of the nanokernel, this is yet to be verified.

and performing a barrier will ensure each core is coherent before the sweep, and that none of the revoker's loads are reordered before the start of the sweep. Due to filter registers, no CPU core can ever write a needs-revoking capability to a memory location. It is therefore impossible for the revoking core to see a value that does not need revoking, while another core sees a value that does. The inverse is possible: another core may overwrite a location that contained a needs-revoking capability with one that does not, and the revoking core may see the stale version. If a location does need revoking, the revoking core will always notice, but might sometimes hit a false positive. The atomic CAS disallows false positives. Unusually, the stored value is the same as the compare value. This is because the stored value will be untagged automatically by the filter registers, and so is effectively different. If the CAS succeeds, the value was not stale. If the CAS fails, another core must have modified the location, but must also have cleaned it. There are significant performance benefits to avoiding atomics; see section 4.4. On MIPS, there is no CAS instruction, and this step is implemented with an LL/SC loop[6].

Apart from two barriers, and the forced context switches, this mechanism is completely concurrent. Other cores are only disturbed by memory bandwidth being consumed and by atomic operations when a needs-revoking capability is found. As the nanokernel already tracks which physical pages are in use, only the in-use subset of physical memory is swept. Furthermore, CHERI includes an instruction to fetch capability tags without bringing other data in from RAM[7], further reducing cache and bandwidth impact if scanning over data not containing capabilities.

The number of filter-register triples a processor should offer is a trade-off between chip space and revocation efficiency. More triples could also effect processor critical path, as they require associative lookup, so a low number is more practical. The current implementation features only one, as this is the easiest to justify being implementable. One is easily sufficient to keep up with the system's memory churn rate, see section 4.4. Sweeping physical space allows reuse of virtual space, giving an exceptionally low amortized cost to recover one virtual byte.

4.3 Cross-process capability exchange

One of the great advantages of being in a single address space is the capacity to share pointers to complex data structures across process boundaries. With a more coarse-grained access-control scheme, this would prove impossible to do efficiently yet securely. With CHERI capabilities, a single write can grant access to complex data structures, with any set of permissions[8]. However, allowing this poses problems of memory ownership and lifetime. Although we may trust our own process to not free data prematurely, no such guarantees can be made about others. Furthermore,

[6]Looping is necessary, as store conditional is allowed to fail for any reason, not just because the location was modified by another core.

[7]This is only possible as the CHERI-MIPS implementation stores tags in a separate block with a separate cache.[67] If tags were interspersed with data, in ECC bits for example, this instruction would be harder to implement.

[8]CHERI-MIPS currently does not implement a 'transitive read-only' permission bit to ensure no writeable capabilities are readable, although it is an experimental feature defined by CHERI. This makes it harder to share pointer-containing structures in a read-only way. This missing permission is implementable using sealing. However, this requires a compartment change to emulate reads with the missing permission bit.

once a capability is granted to another process it is impossible to get it to cease using it, without either revocation or interposition. This limits the guarantees we can make if capabilities are shared.

Rather than not allow sharing, CheriOS instead offers several techniques to make sharing efficient and programmer-friendly. The different methods supplied allow a trade-off between complexity-of-use and performance in different situations.

The memory manager

Neither the CheriOS nanokernel nor microkernel have a notion of either memory ownership or processes. Instead, responsibility for managing resources and providing such abstractions is delegated to less privileged compartments. Users need not interact with these, but the CheriOS C runtime uses them extensively. The entity responsible for managing memory and enforcing resource limits at a page granularity is called the *Memory Manager*. CheriOS instantiates a single memory manager for the entire virtual space. It is a CheriOS design pattern that each manager create its own set of principals, orthogonal to any others, for the resource it manages. CheriOS system services are distrusting, so cannot trust others to maintain a sensible set of principals. The memory manager maintains its own Memory Ownership Principals (MOPs), which have authority to request memory be mapped or unmapped. When a programmer wishes to make any requests of the memory manager, they must provide a MOP with adequate resource limits. A common pattern is that MOPs will correspond 1:1 with processes.

In order to support mutually distrusting applications, the memory manager assumes MOPs distrust each other, so one cannot cause memory used by another to be unmapped prematurely. To achieve this, for every virtual page, the memory manager tracks claim counts per MOP.[9] Tracking is range-based, so claims for large ranges are updated in constant time. A virtual page is only ever freed by the memory manager when every claimant has reduced their claims to 0. This ensures that pages stay mapped if any MOP requires them. When a page has multiple claimants, they all have their resource limits billed.

The memory manager itself is granted a reservation to all virtual memory at start-up; see section 5.2. Reservations are only provided by the memory manager when a virtual page is requested for the first time. MOPs have authority to request that specific virtual pages stay mapped. MOP claims do *not* give authority to access memory (which is governed solely by memory capabilities and reservations). Any principal with a suitable allocation limit may choose to insist a virtual page stays mapped. Other claimers of the range are free to renounce their claims and be refunded their limits. This vastly simplifies the memory manager: it does not track capability flow in the system, only total allocation limits and which pages are currently claimed by which MOP. Not tracking capabilities leaves users free to share memory capabilities in fine-grained ways without the overhead of interacting with the memory manager. Principals do not pass memory capabilities to the memory manager to claim and free pages, only virtual addresses.

[9] There are practical limitations on how many MOPs can claim a particular page. If many MOPs wish to claim a particular page, they must elect a trusted delegate MOP to claim on their behalf.

This is important as the memory manager should never gain access to the memory it manages. Virtual addresses are a good choice, as they are derivable from memory capabilities, which are already being exchanged as pointers. MOPs cannot free another's pages, so no additional authorising capability other than a MOP is strictly necessary. Using virtual addresses avoids having to modify programs to track additional authorising capabilities.

It is important that claims are counts rather than single bit trackers. This avoids confusion when a process is passed a reference to its own data from another process. When the process receives the data, it will put in a second claim (not realising it already has a claim), and, when it is finished, will free that claim. With a naive saturating single-bit tracker, a fault would soon occur. This problem is similar to the one addressed by recursive locks: if a lock is taken twice, the first release should leave it locked.

A MOP handle, like all principal handles in CheriOS, takes the form of a sealed capability. The memory that the MOP points to is a constant size record that tracks which pages the MOP has claimed, and the resource limits for the MOP. The memory manager has the sealing/unsealing capability for MOP handles. It seals any references it passes out, and unseals them on receipt.

MOPs are hierarchical. To create a MOP, another parent MOP must be provided, and give some of its resource limit to the child. The limit is recovered when the child is destroyed. Thus, programmers can create as many MOPs as they desire without increasing their resource limits. The hierarchical nature of MOPs does not mean that the ability to access memory is made hierarchical. Holding a MOP does not grant any authority to request memory capabilities to pages claimed by a MOP. A MOP has only the authority to make requests about the tracking of memory.

It is important to note that a MOP does not necessarily correspond to a particular process, and the MOP hierarchy need not match the process hierarchy. The process manager (the entity in charge of creating a POSIX process abstraction) does, however, create a MOP for every process it creates, rooted under its own. It also tells the memory manager to destroy this MOP when the process exits. In this way, the POSIX model of resources being released when a process exits is implemented, even if the process crashes and memory capabilities become unrecoverable. However, if a programmer wants some memory to outlive a particular process (for recovery purposes, for example), they can simply have another MOP that claims it. MOPs can even be re-parented, so that if a process decides it wants a MOP to have a longer lifetime, it can find another to act as its parent and transfer it. This separation allows for more flexible object lifetimes than binding them to processes, and allows the memory manager to be agnostic to abstractions such as processes. This use case also motivates why MOPs should be hierarchical, and there be a clear split between resource accounting and ability-to-use. The failure of an entity requires some other authorised parent entity to handle cleanup, even if the parent entity has no authority to access the child's resources.

Malloc

The memory-manager interface is page-based, which is inconvenient for programmers, and typically avoided for the sake of efficiency. In order to provide an object-level abstraction, the API to individual malloc implementations is modified to include a new `claim` function. `claim` has similar reference-counting semantics to page-level claiming, but applied to objects. A successful claim means the object can be safely dereferenced. The semantics of `malloc` and `free` are also similarly modified. Calling `malloc` returns a capability to an object, and also places a claim. `claim` will increase the claim count by one. `free` will reduce the claim count by one, and, if this hits zero, then the asynchronous free of the object will take place.

A malloc implementation coalesces claims to multiple objects in a page to only one, reducing the burden on the memory manager, which is a system-wide synchronization point. Typically, each instance of a malloc implementation will utilise a single MOP. When a programmer is passed an object from another process, with untrusted lifetime, they can call `claim` before dereferencing it. If the `claim` succeeds (a failure means the object had been freed), the object is dereferenceable for the lifetime of the claim in the programmer's own malloc, regardless of claims and frees in others. Even if foreign malloc implementations are incorrect, the use of a unique MOP by an application's malloc can ensure correct behaviour for that application.

As with the memory manager, callers to malloc are free to claim objects at a location they cannot access, and malloc is never given memory capabilities[10], thus malloc can be compartmentalised.

Claiming is very similar to reference counting as it is used in garbage collected languages. However, it differs in a few ways. Firstly, the granularity of adjustments to counts are far more coarse grained, and so incur less overhead. Rather than being performed every time a pointer is moved, they are done when objects are acquired by compartments. They are also handled manually, so the shortcomings of reference counting where cyclic structures are involved are avoided. However, also because claim is used manually, a programmer may use it incorrectly.

Guarded instructions

Calling Malloc's `claim` function can still take on the order of hundreds to thousands of cycles. Sometimes a programmer believes an object should already be claimed, and could verify this by reading its state. Also, sometimes, only very few dereferences of a capability are required between a claim and a free (for example, taking a lock). In both these cases, it is advantageous to make tentative access, and explicitly handle unexpected failures in a slow path.

Accessing a freed object throws an exception. Exception mechanisms are somewhat heavyweight, and C does not have good support for them, making their use tedious for a simple check. Instead, we solve this problem with a magic instruction recognised by the nanokernel. This magic instruction is a nop in the CHERI ISA. Whenever a user-handleable exception is triggered, if the

[10]Malloc allows `claim`/`free` to be called on *either* a reservation or a virtual address.

subsequent instruction is the magic nop, the offending instruction is skipped and the exception squashed.

Programmers can make 'safe dereferences' via macros that take a default value for loads if the object has already been freed. These macros ensure correct assembly generation. Here is an example of reading the state from some session object. The state should be set to 'finished' before the object is freed:

```
state_e s = SAFE_LOAD(&session.state, finished); // set s to the value of session.
    state, or 'finished' if that memory has been freed.
if( s == finished) return -1;
```

This pattern is easier to write than using exceptions, and is far more efficient than putting in a claim with either malloc or the memory manager. It is used extensively with shared locks and session handles. Failure to use a guarded load can result in an exception.

User exceptions

The final memory-sharing protection mechanism offered is user exception delivery, which matches closely with copyin/copyout on traditional UNIX kernels. In this case, programmers must manually handle exceptions arising from faulting memory accesses, and must write logic to handle them in a sensible fashion. The exact way that synchronous exceptions are handled and delivered is discussed in section 5.1. More discussion of IO in CheriOS can be found in chapter 6.

Comparison of access techniques

The three techniques offered are meant to trade off between performance and complexity of use.

Claim lends itself best in cases when a single object is to be shared, for a long or indeterminate amount of time. It has constant, but not insignificant, performance overhead, and requires the same programmer burden as malloc's `free` typically does. Failure to use claim correctly can cause memory leaks (as it can in standard C), and freeing too early can result in traps, but never a violation of memory safety.

In comparison, guarded instructions are meant for use when a small number of accesses are being made. They require a small but linear overhead in the extra number of instructions to execute. In the common case of no user attack / error it costs only a single extra nop for each potentially faulting access. It is even quite fast in the uncommon case, requiring only a few dozen cycles to skip an exception. The pattern for their use is very simple, only requiring programmers to specify a default value for unmapped memory. Failure to use a guarded instruction will result in exceptions, that if unhandled, will cause the program to exit.

Lastly, when multiple buffers spanning multiple pages require brief sharing, and many accesses will be made, exceptions should be used manually. This is the case for scatter-gather IO. Claiming individual buffers would incur too high an overhead, and using guarded instructions would generate

too many additional instructions. The requirements on programmers to use exceptions correctly are higher, as they have to handle the clean up of program state that has taken an exception half way through executing. Failure modes are a little more complex, as although it is still impossible to violate memory safety, bad exception handling logic can nevertheless introduce vulnerabilities. Programmers should rarely have to use exceptions explicitly, but specialised libraries like the socket library use it. If freed data is sent along a socket, the socket library handles the exception and closes the socket. The process sees an error return code when calling the socket operation.

The three methods offered were chosen as they were both low-level and did not require cooperation from possibly adversarial compartments.

4.4 Evaluation

The strategies employed to provide temporal safety require both increased interaction with the OS and spend time making memory re-available via revocation. Here we look at the costs of both, and how they impact the running system. The costs of CHERI's spatial safety are better discussed in existing work.[108][37] As with all our benchmarks, we run CheriOS on CHERI-MIPS, synthesised on a Stratix IV FPGA with a single core, 6-stage in-order pipeline, at 100MHz, 1GB DDR2 RAM, and 256KiB LLC configuration. All programs were compiled using LLVM version 7, modified to support CheriOS.

Revocation

The dominating cost of revocation is sweeping. Here we evaluate how long sweeps take under different conditions, and how they affect a running system.

Cost of a sweep

These benchmarks show the cost of the revocation sweep in isolation. The system was kept mostly idle, but a few outliers were generated due to background activity. To artificially increase memory consumption, a benchmarking program allocates memory on its heap and then forces a revocation, but otherwise does no work.

CheriOS relies on being able to delay revocation, waiting until large virtual ranges become eligible for collection. We hypothesise the size of the revoked range should have no bearing on performance, so such a delay will be of benefit. To measure this, the virtual range being recovered is varied while keeping the physical range swept over (approximately 200MB) as constant as possible. The physical range varied less than 1% between samples. Figure 4.5 shows the result of this experiment. As expected, there is no noticeable difference in the time taken to perform a revocation. The difference between the time to revoke 4KB and 128GB is approximately 0.38%. Although a small overhead is incurred for modifying PTEs, the maximum number of pagetables needing modification is bounded at five for a three level table, i.e., the root and a first and last

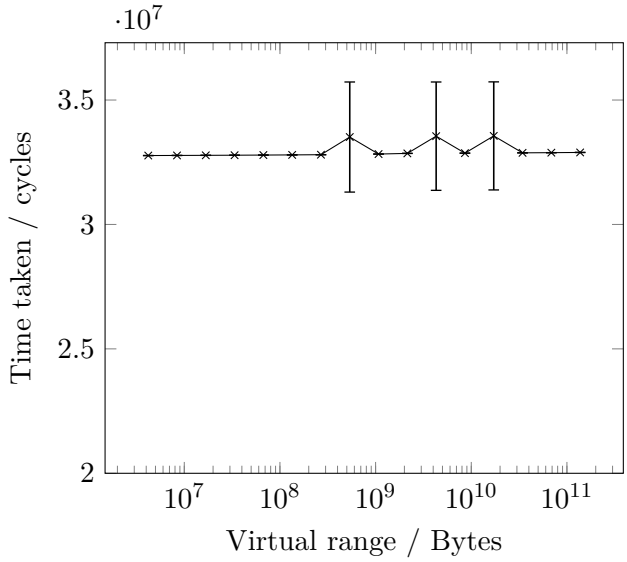


Figure 4.5: *Revoke sweep time dependence on virtual range being revoked. Error bars indicate one standard deviation. Errors caused by hard-to-control system load, such as IRQ traffic.*

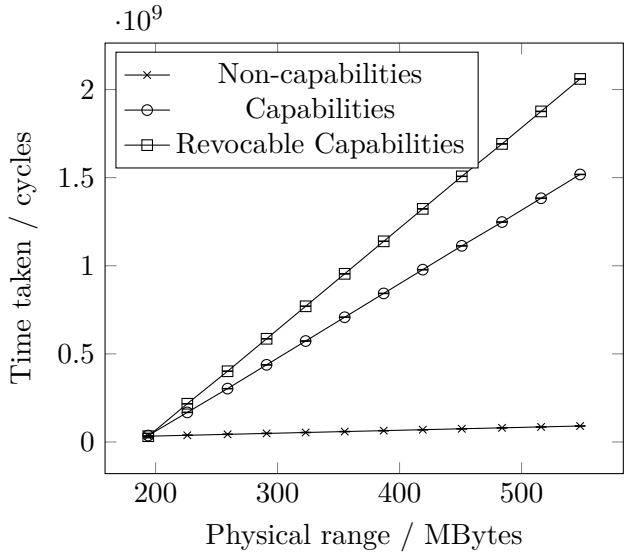


Figure 4.6: *Revoke sweep time dependence on physical range swept over. Error bars indicate one standard deviation and are mostly invisible.*

table for the other two levels. The dominating step in revocation, sweeping memory, does not depend on the virtual range being revoked.

The physical range needing to be swept reflects how much memory is in system-wise active use. To artificially vary this, the dummy program allocates differently sized objects on its heap before triggering a revocation. The virtual range being revoked is kept constant, at roughly 4MB. Memory contents have a major impact on revocation time, so we fill the contents of the heap with three datatypes: non-capability data (all zeros), capabilities (but not those that need revoking), and capabilities that need revoking. After each revocation, we refresh the heap with these values,

Type	K Cycles / MB	Non-Capabilities Relative	Capabilities Relative
Non-Capabilities	164	-	-
Capabilities	4185	25.5	-
Revocable	5727	34.9	1.4

Table 4.1: *Cost per megabyte swept during revocation for different data types.*

in case revocation changed them. Figure 4.6 shows how revocation sweep costs vary as the physical range swept increases. The cost for real memory contents would be somewhere in the resulting cone. The physical range is the system-wide load, not just the benchmarking program's heap. The first point (where all lines intersect) represents where the heap is empty. Please note, it is highly unlikely to have memory containing only capabilities that need revoking, or even only capabilities. Programs tend to have some non-pointer data, and revocable capabilities can be relatively rare as pages are unmapped before ever being swept. It was observed that revocation sweeps running after a program exits, a common trigger, would find revocable capabilities in only the single digits.

As expected, time varies linearly with the size of physical memory swept, for any of the heap loads. It should be apparent why the revocation logic tries to only load tags. The cost of sweeping over capabilities or revocable capabilities is ∼26 and ∼35 times (see figure 4.1) that of untagged data, respectively. It is worth noting that the benchmarking CHERI-MIPS processor features neither a pre-fetcher nor pre-fetch instructions. The sweep is therefore likely to regularly fault (memory is much larger than total cache space), even though the access pattern is very predictable. This accounts for some of the discrepancy. The access pattern is a trivial stride, a non-experimental version of CHERI-MIPS would probably perform much better.

The actual proportion of tagged to untagged data is heavily workload dependent, and is impacted by every program running in the system. The CHERIvoke[138] work includes a survey of the proportion of pages (not capability-sized words) that contain any capabilities at all, across several applications. The proportions range from less than 1% to up to 95%, and include many values in-between. Importantly, large caches allocated by the OS, such as those for files, contain no capabilities, so will take advantage of the much faster sweep. Pointer-intensive programs will move the system closer to the middle line, but you would never expect to reach it. An adversarial program may cause revocation times approaching the greatest gradient, but CheriOS does not seek to address denial of service.

Concurrent sweeping

One benefit to delaying sweeping almost indefinitely, which we achieve using MMU-based revocation, is we can do so when the system would otherwise be idle. Default behaviour in CheriOS is to make revocation a very low priority task, unless virtual memory is about to run out, and as it happens so infrequently it will rarely interfere with system performance. However, we might imagine a use case where the system was under constant maximum load, and sweeping

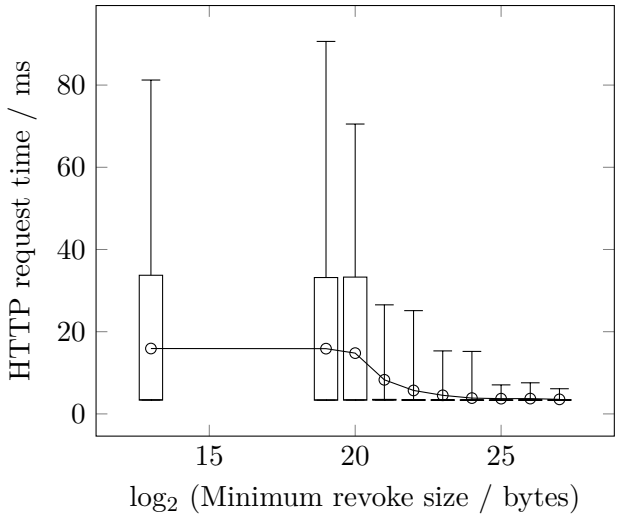


Figure 4.7: *Revocation sweep's effect on HTTP response times. The median response time always was too close to the lower quartile to show distinctly, as the distribution is heavily skewed in only one direction by revocation events. The whisker shows one standard deviation from the mean, and the line shows the mean response time.*

was necessarily concurrent. We might also recover very sub-optimal regions of virtual memory, due to either only supporting a small virtual space, or high fragmentation. For the purposes of this test, the revoker has a high priority to ensure it runs concurrently, and the minimum virtual range eligible for revocation is varied (with values much smaller than would normally be used), so it triggers more often. Regardless of the minimum range revoked, the revoker will always select the largest range available as a revocation target.

A HTTP server (NGINX) under heavy load is used to drive memory consumption. A similar setup is used in chapter 6, where the performance of CheriOS is examined more closely in comparison to contemporary systems. Here it serves only to illustrate the rate at which memory may realistically be consumed, and provide a non-trivial workload. Each test was made of 20 000 requests for a 1KB file.

Figure 4.7 shows how long HTTP requests took to serve as the minimum revocation threshold was increased. The last point has no revocation sweeps occur during the benchmark, so is indicative of how fast NGINX runs without any revocation, although revocation would inevitably occur at some point if the test ran longer.

Figure 4.8 shows how many revocation sweeps occurred during each benchmark, and how much virtual memory was recovered as a percentage of what was freed. Setting the revocation threshold below 1MB has no effect as the minimum memory size exchanged between the OS and malloc implementations is of this size. After this point, revocation events drop sharply, and similarly so does the overhead. The memory recovered, however, stays relatively high, never dropping below 60% when there were any revocation passes.

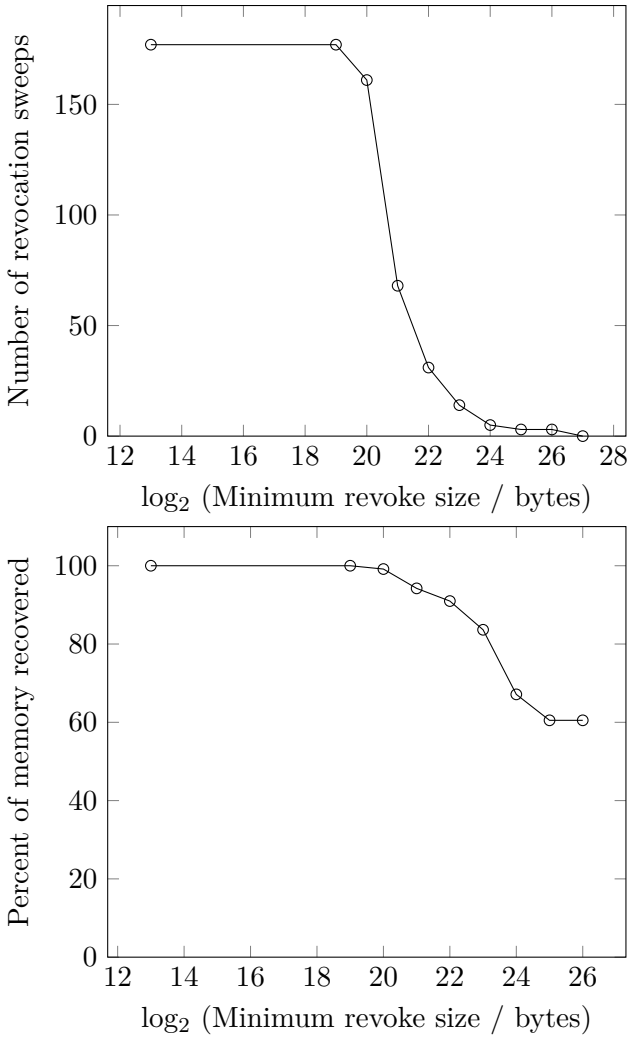


Figure 4.8: *Revocation sweep's effect on revocation efficacy. Last point omitted as no sweeps ran.*

An interesting estimation is the minimum revocation frequency possible, and what overhead this would incur. The minimum frequency depends on the rate the system would free memory (r), memory recovered per sweep (m), and the time it takes to perform a sweep (t).

$$overhead = (t * r)/m$$

Roughly, the system-wide free rate of the previous benchmark was 355MB over 71s, or $r = 5MB/s$, and sweeps took roughly 1s each, so $t = 1s$. Harder to estimate is the maximum recoverable virtual space, as this depends on fragmentation. The current CheriOS pagetable hierarchy allows for 40bits of virtual space. Only 1GB of the 1TB of virtual memory can ever be in use (the size of physical memory), so we could potentially recover ∼1TB, but we will probably recover less due to fragmentation. Estimating we can recover an 8th of memory, this gives $m = 128GB$ and an overhead of as low as 0.004%. For the current workload, such a recovery

is possible, as no applications have objects that are neither permanent (a part of the 1GB in use portion of memory), nor have lifetimes of more than the order of milliseconds. This will not be the case for other applications, but a larger virtual space can offset fragmentation.

Slinky stacks

Consuming stack segments generates memory churn, which the OS handles identically to heap churn, and the combined impact was measured when benchmarking revocation. Here we will look more closely at the independent cost of slinky stacks.

Programmers can optionally turn slinky stacks off, if the extra performance is desired, and they feel the application handles no sensitive data or powerful authority. If a program has not been reasoned about, the default is enabling them, as disabling them allows temporal safety vulnerabilities. OS compartments that handle sensitive data, or handle communication between other compartments, always run with slinky stacks on. For the following benchmark, to measure impact, we attempt to turn all slinky stacks off or on system-wide, even when it would not be appropriate for security. Some early loaded programs, like the microkernel and page allocator, currently never run with slinky stacks due to bootstrapping issues. The kernel currently has no stack behaviour that cannot be proven safe. Time spent in both of these compartments is relatively small for the workload we perform, so should not noticeably impact results.

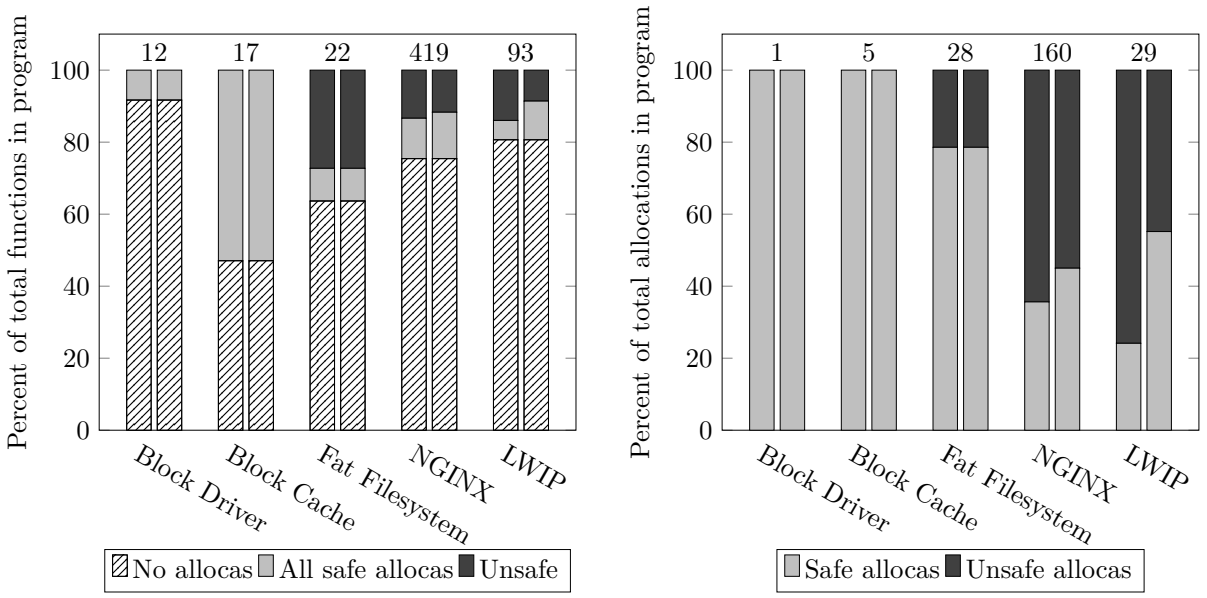
There are two metrics that impact the cost of slinky stacks: how many objects there are that require the unsafe stack, and how many functions require any at all. The first corresponds to how fast segments are consumed, and the second to how fragmented stacks become. In the end, all that matters is dynamic cost, but it is informative to see static metrics.

The corpus of programs we analyse are all those on the path for the HTTP benchmark, that is: the block driver, block cache, filesystem, NGINX, and LWIP. A temporal error of any of their stacks might lead to exploits that could compromise data being sent and received.

Static analysis

The temporal-safety analysis pass runs relatively late in the LLVM pipeline. By the time it does, LLVM has already attempted to promote `allocas` to registers, so there are relatively few `allocas` remaining. Using the escape analysis described in section 4.1, we attempt to further partition these into safe and unsafe. A few `allocas` were manually tagged with the safe keyword: for example, references passed to the nanokernel are considered safe.

Figure 4.9a shows the relative distribution of functions without `allocas`, with only safe `allocas`, and with any unsafe `allocas`. Each program is analysed twice, once with no static expansion at call sites (left bars) and once with an expansion limit of 64 (right bars). The total number of functions in each program is labelled atop each bar. Figure 4.9b shows the relative distribution of safe and unsafe `allocas`. The total number of `allocas` analysed is again printed atop the bar.



(a) *Function granularity.* **(b)** *Allocation granularity.*

Figure 4.9: *Temporal safety analysis at function and allocation granularities. Right bars are with inter-procedural analysis. Labels at the top show total absolute numbers.*

As the figures show, some of the simpler programs are proven to have no unsafe allocations at all, and so stack temporal safety is achieved without any overhead. CheriOS has an untrusting calling convention (see chapter 5) that allows compartments to safely call untrusted functions. The compiler requires slinky stacks to allocate continuations for use by the untrusting calling convention, so even with no explicit unsafe allocations, a distrusting program uses its unsafe stack anyway.

Drilling into the programs themselves, we can see what kind of patterns result in unsafe allocations. Every program that had unsafe functions had an unsafe main; this was due to the common practice of allocation structures containing global state on the stack in main. Standard libraries were also to blame: CheriOS libraries, such as those for file handling, pass pointers to the stack across compartment boundaries to return values. These could be eliminated at an ABI level using more return registers. Standard libraries account for the majority of unsafe functions in every program but NGINX. NGINX routinely manipulates stack-allocated strings, such as those for logs and URLs. Combined with a heavy use of function pointers (which hamper analysis), many unsafe functions result.

We can also see the cases where inter-procedural analysis was helpful. LWIP often returns integers via stack pointers, such as for extracting length and IP fields. These are obviously safe in most usages, but require static expansion to prove. NGINX's parsing functions that target on-stack string buffers are also better handled with inter-procedural analysis, when not called indirectly.

Program	Unsafe depth	Churn rate / MB·s^{-1}
NGINX	9	1.87
LWIP	4	0.06
FAT Filesystem	2	0.03
Block Cache	3	0.25
Block Driver	1	0.01
Total	-	2.21

Table 4.2: *Unsafe stack usage for different programs.*

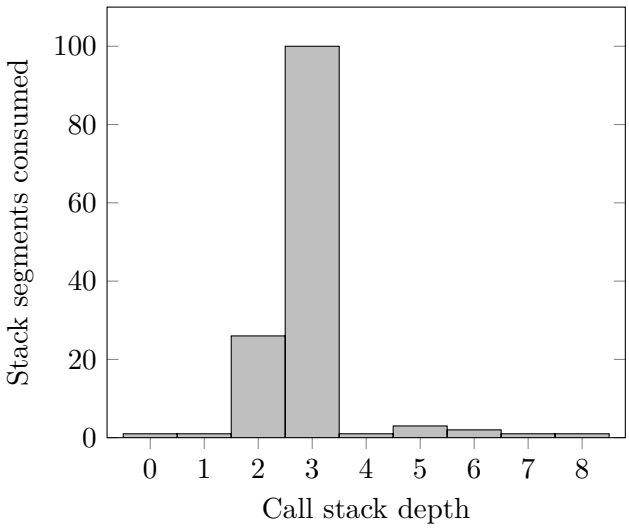


Figure 4.10: *NGINX's stack segment consumptions across unsafe depths.*

Dynamic cost

To measure dynamic cost, the same 20 000 HTTP request workload used to benchmark revocation was employed. At the end of the test, the total number of stack segments used, and the maximum depth reached, was recorded. Table 4.2 shows both maximum depth and memory churn rate. A conventional contiguous stack would have zero memory churn, and a depth of one segment. However, normal stacks sacrifice temporal safety of stack allocations. Other compartments may gain access to objects they should not, and possibly control flow could be hijacked within a compartment.

Note that stack segments are discarded before they are full. Therefore, this rate is larger than the allocation rate of unsafe stack objects. For the purposes of calculating churn rate, stack segments in use at test-end were considered full, in order to over-approximate. Nothing currently compiled for CheriOS utilises dynamic unsafe allocations, thus the number of segments in use is equal to the maximum unsafe depth, plus one. For example, the block driver never uses the unsafe stack, and NGINX uses at most 8 unsafe segments at once. If we compare the system-wide stack churn rate of 2.21MB/s to the total memory churn rate of 5MB/s measured in section 4.4,

	Request Times / ms		
Stack Type	Mean	Median	Standard Deviation
Normal stack	3.306±0.014	3.202	2.041
Slinkly stacks	3.438±0.017	3.301	2.351

Overhead of Slinky Stacks		
Mean	Median	Standard Deviation
3.99%±0.68%	3.09%	95.81%

Table 4.3: *Request times for 1KB file with and without slinky stacks.*

we can see that temporally-safe stacks account for much of revocation pressure. However, due to design, they leave very compact footprints, and so interact well with revocation.

Note that a depth of 9 does not imply that we are actively using 9 segments. To illustrate this, figure 4.10 shows a histogram of how many stack segments were consumed at each unsafe depth when running NGINX. Note how the vast majority occur at depth 2 and 3. This means that most transitions were between segments 1 and 2, and 2 and 3. Thus, only 3 stack segments were commonly used. Looking at the actively used segments paints a better picture of slinky stack's spatial locality than by just looking at the maximum unsafe depth.

We also measure the effect of globally turning safe stacks on or off. This will not include any overhead for revocation sweeps, as unlike in section 4.4, the revocation threshold has been set to a sensible value, running on a time-scale of minutes. Table 4.3 shows the mean and median HTTP response times for the system with slinky stacks enabled and disabled. Both overheads make the technique very competitive with other schemes for temporal safety. Although the static analysis used was CHERI specific, slinky stacks could be deployed on OSs other than CheriOS.

4.5 Related work

Protecting stacks

Here we present some related techniques, with a focus on CHERI-based approaches, for enforcing both spatial and temporal safety on the stack. Most work utilising capabilities for temporal safety does so by limiting capability propagation. This starkly contrasts CheriOS's slinky stacks that allow any propagation, but must then recover capabilities via revocation. Obviously, revocation incurs a cost, but the added flexibility is crucial in allowing compartments to share capabilities between themselves.

Local/Global capabilities

One CHERI capability feature not utilised in this thesis is the presence of a 'local' bit. Capabilities with the bit not set are global. A local capability is storable only via a capability with the 'store local' permission. This allows controlling propagation of local capabilities by limiting where they are storable (a property of the reference, not location). In related work they have been utilised to provide temporally safe stacks, or something close.

Reasoning About a Machine with Local Capabilities Skorstengaard et al.[120] provide encapsulation of stack frames using only local/global capabilities. The general idea is to ensure a callee cannot keep hold of capabilities they should not by only allowing stack capabilities to be stored to the stack, and then regularly zeroing the stack. This is achieved by making the only available capability with the 'store local' permission the stack capability, and making stack capabilities local capabilities. On call/return, functions will zero the *entire* unused region of the stack, ensuring no stack capabilities are stashed there. When a function is called, the caller restricts the bounds of the stack so the caller's in-use portion cannot be accessed by the callee. The callee will only have access to arguments, and an initially all-zero stack. Stack pointers are stash-able neither on the stack (it will be later zeroed), or the heap (it does not have the store-local permission).

This does not actually enforce temporal safety, but rather stops unintended information and capabilities leaking between stack frames. Dangling stack pointers are still dereferencable, for example if returned, but the objects they point to are zeroed between call/return. This protects information leaks, but still allows unpredictable and potentially exploitable effects if capabilities are used outside their lifetime. Another downside, apart from the cost of zeroing megabytes of stack twice for every function call, is that stack pointers cannot reside on the heap, in global storage, or be passed to other threads. Also, the OS must guarantee only the stack capability has the 'store local' permission. With mutual distrust, such as CheriOS desires, this is unenforceable, and the scheme cannot work.

The scheme is also subtly flawed. They assume that because they restrict the bounds of the stack-capability on call, it is impossible for a callee to access the caller's stack frame. This is not true; the caller may provide additional arguments with stack capabilities, and compilers often generate such arguments. This is not obviously a problem, as the caller intends to pass this section of their stack (it should contain no sensitive information), but it does give a way for the callee to stash local capabilities for use in future invocations (a good choice would be in padding around objects as the caller would not notice any strange behaviour). Stripping 'store local' permissions on arguments passed does not resolve the issue, as stack-resident data structures walked by the callee can contain further capabilities to the caller's stack. The compiler also cannot zero user objects passed by reference, as this fundamentally breaks C.

Temporal safety for stack allocated memory on capability machines Tsampas et al.[128] propose a similar solution of limiting propagation of stack capabilities, but rather than a

single local bit they have multiple 'levels' a capability may have. A capability of a level 'X' is only storable via a capability with authority to store a level greater than or equal to X. Local/Global capabilities are a 1-bit version of this, level 0 is global, and level 1 is local. They achieve temporal safety by assigning every call frame a level, with the first stack frame having level 0, the next level 1 and so forth. Due to storing rules, an object cannot contain a reference to another object that would outlive it, so if no out-of-lifetime references are held, it is impossible to find any by making dereferences. Registers are cleaned between function calls to ensure out-of-lifetime references are not passed as return arguments. Like with the previous technique, this prohibits storing stack pointers to the heap, global storage, or thread-local memory. Crucially, it also disallows storing a stack capability to a caller's frame, as broke the previous scheme. Obviously, this further reduces flexibility, but as far as I can see is sound in comparison.

This technique does not require zeroing the entire stack. It does, however, require zeroing stack frames before use, and filtering the register file on function return. It also requires a trusted third party to elevate the stack capability's level, so cannot support mutual distrust as easily. Spare bits are already quite sparse in CHERI capabilities; there are not 16 (the suggested amount) spare in the current format, so it may be infeasible to require that many.

Linear capabilities

A linear capability is one that can only be moved, not copied. This requires modification to hardware to enforce, and is challenging for both software and hardware architects. However, they offer a very promising way of ensuring temporal safety. If a linear capability is 'given back' to an allocator, then we can be sure that no more references exist. Although they have not been demonstrated with CHERI, they are an active area of research. Linear capabilities are split rather than just subset, and can be spliced back together as well. If a linear capability starts out not aliasing with another, it can never alias with another.

StkTokens Work by Skorstengaard et al.[121] uses linear capabilities to enforce both isolation between functions, control flow integrity, and temporal safety using linear capabilities. To do so, they make the stack a linear capability. Callers split off a part of the stack for the callee, and the callee must return this part to be spliced onto the stack in order to return. The caller can ensure the callee can no longer access the stack because they cannot have both kept a copy and returned it, and the callee can trust the caller cannot access their frame during use as the linear capability provided must be unique.

This scheme can support mutual distrust and provide temporal safety with potentially very low costs. However, it comes with all the restrictions that linear capabilities come with. Every linear load is in fact an atomic swap, as the source needs clearing. If the update were non-atomic, multiple cores could conspire to duplicate linear capabilities. Instructions also must clear their inputs, which means register files require more write ports. Programmers are severely limited in the data structures they can build, and have to make stylistic changes, as most uses of a variable will make it dead. Compilers will also have to be restructured. Currently, the challenge with linear capabilities is not devising how they may be useful, but in overcoming

their implementation difficulties. Making them work practically with existing languages, and implementing them microarchitecturally, are outstanding problems.

Spatial safety on the stack

Here we briefly include some non-capability based techniques that are in active use (hardware enforced memory-capabilities are not) as a point of comparison. A more in depth evaluation on the merits of CHERI vs other techniques for spatial safety, such as all the techniques using fat pointers, can be found in other works.[37][108]

Tagged hardware with metadata policies PUMP[42][41] accelerates dynamically-configured policy checking for tagged machines with n-bit tags. This contrasts CHERI, which has a single tag bit, and semantics are architecturally defined.

Roessler and DeHon[109] use PUMP to define policies that achieve almost complete spatial safety on the stack. Their approach is reminiscent of colouring: they assign the same tag to pointers and the memory they should reference, and policies check tags match. If the object is accessed relative to the stack pointer, the colour is instead encoded in the instruction.

Their first scheme uses only two tags: one for return addresses, and another for stack data. This offers control flow integrity for a small 1.2% runtime overhead. Their more complicated schemes assign tags that are pairs of an object ID, and either a static function ID, or a dynamic call depth. These incur overheads of 5.7% and 4.5%, respectively.

Like other colouring schemes, the re-use of colours creates both temporal and safety vulnerabilities. With all their policies, temporal safety is not achieved where colours are reused. This is slightly stronger than the naive bounds-only CHERI approach (not using slinky stacks), but much weaker than CheriOS's approach which does offer temporal safety. However, spatially, it is slightly weaker than even a naive CHERI approach. CHERI stores bounds in the reference, colouring stores bounds in the object. An out-of-lifetime CHERI capability is therefore still spatially safe even if temporally unsafe. An out-of-lifetime coloured pointer becomes spatially unsafe as well, because its colour will almost certainly be re-used for an object in a different location.

Even with reuse, the number of tags needed is relatively high. The paper reports an average of ∼5000 or ∼1000 tags for their benchmarks (13 or 10 bits), depending on policy. Bloating pointers this much is no worse than CHERI, but the extra tag memory is significantly larger than what CHERI requires.

Stack canaries Stack canaries are a common technique to detect, but not prevent, buffer overflows. The name comes from the use of canaries in mines to detect dangerous gases. If the canary dies, it is time to abandon the mine. Stack canaries are objects placed in-between others, and are given a special value that should never change. One is generally placed between programmer objects and important spill slots, such as the return address. This value is checked,

generally just before returning from a function, and if it has changed then corruption has occurred. Just as with a mine canary, if the stack canary is dead, it is time to abandon the process.

Stack canaries stop the simplest of spatial violations: ones were memory is being stepped through, think `memcpy`, `strcpy` etc. Historically, these adjacent attacks were most common. However, likely as a response to prevention techniques, exploits have moved on to use gadgets that use larger offsets that skip over canaries. Microsoft reports[26] that, since 2006, the proportion of attacks using adjacent overflows has gone from 90% to just 25%. Stack canaries are obviously not sufficient to stop a modern attacker.

Shadow stacks A shadow stack is a separate stack that contains sensitive objects. LLVM calls its implementation 'SafeStack'[75], which is a part of its Code-Pointer Integrity (CPI) project. LLVM employs a shadow stack for important objects, and those that cannot be used in unsafe ways, putting dangerous objects on the main stack. This still allows spatial violations, but limits which objects are corruptible by separating valuable targets from dangerous objects, such as arrays. Like with stack canaries, this only stops adjacent attacks, and even worse will still allow adjacent attacks on some objects. It offers some defence in depth, but cannot compare to true spatial safety. For what they offer, shadow stacks are quite expensive. Dang et al.[36] say that traditional shadow stacks cost on the order of a 10% overhead. The original CPI paper[75] puts the cost of safe stacks at 8.4%, when putting as many objects as possible on the shadow stack, or 1.9% if just protecting the return address.

Revocation

BOGO BOGO[100] notes that the tables in Intel MPX (which contain bounds information for every pointer in a process) can be scanned on every free to remove any dangling references. This is effectively a revocation sweep on every call to `free`, although the MPX tables (which contain only bounds) are swept, not process memory itself. This would of course be prohibitive, as metadata scales with number of pointers, so BOGO divides memory into hot and cold sets. Only hot memory is scanned on free, and cold pages are unmapped so their use causes a fault. Cold pages are then dealt with lazily, although this lazy handling means that memory cannot be reallocated until the queue is dealt with. Thus, like with much temporal-safety work, there is a trade-off between non-reuse and performance.

Even with lazy scanning in use, an average overhead of 50% is reported, with some benchmarks being as much as 14x slower. This is *on top* of the already costly use of MPX, which has seen little success due to its cost with respect to both performance and memory.[100] These costs are far above what is commonly accepted for adoption. In comparison, CheriOS has much lower performance costs. However, at present, CheriOS's handling of the heap is insufficient for many workloads due to fragmentation, and fixing this is ongoing research.

CHERIvoke CHERIvoke[138] provides fine-grained revocation, at the granularity of individual objects, for CHERI capabilities. Like the revocation offered by CheriOS, it is sweep based, and

so is costly, but this cost is amortised by the revocation of many capabilities at once. Which capabilities need revocation is stored in a memory shadow map. When an object is freed, the map of the object is painted. At the next revocation epoch, the memory can be re-used. This map is too large be checked on every store, and so no filter register solution like that of CheriOS is possible. Instead, several sweeps are made using pagetable protection bits to track which pages may still be dirty, and stopping the world for the last sweep of remaining dirty pages.

Batching revocation still wastes some memory short term, but this can be kept quite low and is a trade-off with performance. Setting the memory overhead at 25%, the paper predicted a performance overhead of 4.7% on SPEC benchmarks (worst case 50%).

The biggest adoption hurdle of fine-grained revocation for CheriOS is the difference in scope. CheriBSD divides processes by address space (like nearly all UNIX variants), and capabilities cannot be shared between them. Therefore, stopping the world for a malloc implementation means only stopping a single process, and the working set of that process is all that needs sweeping during the stop. In contrast, CheriOS processes share capabilities. This means stopping the world means stopping *every* process, and stopping for long enough to scan the entire system. Stopping a single process is mostly acceptable; processes are time-shared anyway, and expect to be suspended for other processes. Stopping entire systems, however, is very undesirable, especially if this means interrupts cannot be responded to in the interim. The cost of a complete system stop increases as systems get larger. The advantage of CHERIvoke style sweeping is proper handling of the heap, while CheriOS's currently fragments. It may be that the optimum solution is to use filter registers wherever possible (such as for slinky stacks and for revocation after program exit), and a technique like that of CHERIvoke for the heap.

4.6 Conclusion

CheriOS demonstrates system-wise spatial safety by utilising bounded CHERI capabilities, and temporal safety via selective non-reuse. Non-reuse is even achieved on the stack, utilising a novel stack structure, a slinky stack. When non-reuse fails due to exhausting the address space, CheriOS uses a hardware-accelerated revocation mechanism to ensure reuse is still safe. Although very costly, utilising the MMU, this sweep has its cost amortised over several reuses of physical memory.

Because it can be made safe to do so, CheriOS allows sharing memory objects between distrusting libraries, or even processes. Users can `claim` an object, ensuring future use is safe, or access it via a guarded method that will return a default value if the object no longer exists. We will see in future chapters how these mechanisms are useful for creating performant interfaces, even between distrusting compartments.

Chapter 5

Mutual distrust

In CheriOS, every compartment is optionally isolated from every other in the system. The nanokernel is not considered a compartment for this purpose, as while it is isolated from the rest of the system, it also has complete authority over it. CheriOS system services, such as drivers and filesystems, are built out of at least one compartment, and often more. Applications are designed by users, and are also composed of compartments. As is conventional, system services assume that application compartments are malicious, and so opt to be protected from them. However, less conventionally, application compartments can also assume system services are malicious. This is mutual distrust.

It may not be unreasonable to suggest that such bad actors actually are present in a system. Consider, for example, situations where the owner of the system is different from the user, as is the case with cloud computing. System maintainers may be motivated to spy on their users, either for personal gain, or on the behest of authorities. However, the true reason we have users assume system services are malicious is that even the best-intentioned system architects will always make mistakes. It is unreasonable to expect that large systems be maintained that are completely bug free. Even formally proven systems can have weak specifications, and have limitations in which parts of the system are considered in scope of the proof. CheriOS programs can start by assuming the worst: every other compartment, other than the current, is doing its utmost, potentially in collusion with others, to steal or disrupt program state.

In the CheriOS attacker model, each protected compartment considers all others it does not explicitly trust as adversarial compartments. This protects the system from escalation of compromise between compartments. An adversary can inject arbitrary machine code and data into adversarial compartments. Adversarial compartments can include privileged compartments, such as the scheduler or memory manager. The result of code injection is not allowed to directly change the state of the protected compartment (integrity), or reveal its contents (confidentiality). Control flow (the path taken during execution) of a CPU running in a protected compartment is not allowed to be disrupted from legal flows by an attack. Control flow in a compartment may alter due to an attack; for example, if the application is made to handle an error due to an attacker making a system service unresponsive. Although such a flow would not occur without the attack, it would be still be an intended flow by the programmer who wrote the code for

the compartment. Attackers are also allowed to introduce faults that stop compartments from functioning (like unmapping memory associated with them), but these faults must either be handleable, or completely stop any further progress. In either case, integrity and confidentially must never be violated.

Some problems are considered out of scope. CheriOS does not handle information leaking via side channels (such as those that arise from cache-timing attacks), although it should be noted that CHERI capabilities are secret-free, and so, even with a very powerful read-only side channel, attacker potential is still limited. Furthermore, CheriOS cannot guarantee that OS services will be responsive, or perform their intended function correctly. Because of this, denial-of-service attacks are unsolved by CheriOS. Even if applications have been promised otherwise, it is the OS's prerogative to at any point take back resources from the application, whether that be memory, CPU time, or file access. However, CheriOS will guarantee that the result of doing this will either be handleable, or cleanly kill the program, without compromising the other security guarantees CheriOS offers.

In this chapter we cover the features offered all the way from the nanokernel, which offers a simple set of primitive capability types, through to userspace, which uses these to construct secure compartments offering high-level guarantees of integrity, confidentially, authenticity, and control-flow-integrity. We evaluate the performance implications of supporting mutual distrust, and also compare to existing work.

5.1 Domain transitions

Different layers of the system in CheriOS have different abstractions for a unit of flow control. The nanokernel offers virtual CPU contexts, which are wrapped by the microkernel to offer scheduler activations, which are utilised by OS services and libraries to create threads. There are two major types of transition made in CheriOS. The first kind of domain transition is context switching between the scheduler activations managed by the microkernel, where communication is achieved via message passing. This is detailed in section 5.1, and section 5.1 shows how it composes with interrupts. Although this alone would allow the building of compartmentalised software, using cross-process communication at every compartment boundary would be prohibitive both in terms of performance, and in the complexity of modifying tool-chains to take existing C programs and split them into multiple processes. CheriOS tries to encourage compartmentalisation on fine-grained boundaries, even as far as on every function call.

The second kind of domain transition is made by a thread moving between mutually-distrusting compartments within a process. This naturally aligns with boundaries that programmers create when they write software. Threads can move across multiple orthogonal compartments, and need to ensure that neither capabilities nor information leak between these compartments. Recall from section 3.4 that CheriOS threads are a control-flow abstraction managed by the process manager, and are built on top of scheduler activations. We will call the accessible state a thread has in a particular compartment a domain, in order to not confuse it with the compartment itself, which may allow many threads to execute in it (potentially with slightly different domains).

Existing C programs expect to be able to have multiple threads executing the same function, and concurrently use the same program objects. Therefore, simpler compartmentalisation models where only up to a single thread is executing in a compartment were not considered, as they would not support this behaviour.

Importantly, the microkernel (or some other third party) cannot be relied upon to make domain transitions, as the microkernel is itself an untrusted compartment. As such, we identify two driving principles when designing the transition mechanism:

- CHERI primitives should be the only enforcement mechanism used.
- Compartments should be enforcing the parts of the ABI (application binary interface) that are required for their own security, even if this might mean duplicating effort.

This split of enforcement between both sides of a transition will become apparent when discussing the calling convention developed for CheriOS in section 5.1). How user exceptions are delivered to a domain is covered in section 5.1.

Context switch

The microkernel is in charge of deciding which user programs get to run, and for how long. This does not mean it should be granted the privilege to spy on or alter them. In order to allow the microkernel to switch between a number of contexts, without revealing their state, the nanokernel offers virtual 'CPU contexts'. CPU context handles are sealed CHERI capabilities to a record containing the saved register file (which includes capabilities) of a CPU. Due to being sealed, none of the state is visible outside the nanokernel. Only registers are saved. Any other state required to provide a higher-level abstraction of an object activation (such as a kernel stack, message queue, scheduler state) is handled by the microkernel. The microkernel will create a virtual CPU for each object activation, in order to save a register file.

The nanokernel offers an invocable capability to load a context provided as an argument. The context handle is then unsealed by the nanokernel and used to restore CPU state. The context calling switch is saved such that, upon being restored, the call looks like a no-op from the caller's pointer of view.

The nanokernel also offers a capability to create a new CPU context from a specified starting register file. The microkernel can create a CPU context with any capabilities it has, however, this state is copied into a reservation, which the microkernel must also provide. Reservations guarantee exclusive access to memory (see section 5.2), so the microkernel will be unable to obtain a capability to memory used for the context. This allows the microkernel to decide the state that a CPU starts in, and where to store this state, but never be able to directly access it. Initially, every capability available to a CPU context must have been granted by the microkernel. For new contexts to obtain unique capabilities (e.g. memory not accessible to the microkernel), reservations are used again. Code can attest contexts reach a safe state with the use

of foundations (section 5.3), allowing applications to tell whether the microkernel has interfered with their intended initial state. An example interference might be to add in extra logic that leaks capabilities by writing them into a buffer visible to the microkernel.

Asynchronous exceptions

When an asynchronous exception (such as a timer or hardware interrupt) is triggered, a user's context will need to be preempted, and control transferred to the microkernel. However, to protect user programs, we require that neither the state of the user's context on interrupt, nor the state that will be restored on resume, be made available to the microkernel.

This requirement precludes allowing the microkernel to install its own architectural exception vectors, which would otherwise allow the microkernel access to the interrupted code's capability registers. Instead, the nanokernel handles all architectural exceptions. MIPS does not have a hardware-pagetable walker, and so this is placed in the nanokernel[1]. On an exception (apart from a TLB miss), the nanokernel will save the preempted CPU context and then load a pre-registered exception context (which in CheriOS is the microkernel), giving the exception context no access to the preempted state. The nanokernel offers a capability to the microkernel to set the exception context for each physical CPU core to any virtual CPU context handle. An exception in an exception context that would require switching to the same context again is therefore unrecoverable; a machine restart or rescue by another core would be required to make any further progress.

The nanokernel also offers limited access to information about why an exception happened. For example, if a pagefault occurred, the page number is revealed. The exception context invokes nanokernel capabilities to query which exception occurred, and in a limited fashion why.

Synchronous exceptions, such as those that do reveal program state, are only propagated if the user cannot handle them themselves. Even then, none of the user's capabilities are ever made available to the microkernel. Synchronous exceptions are covered in section 5.1.

This mechanism completely hides user program state from the microkernel. However, it does require two complete register-file switches for an exception. One from the victim context to the microkernel context, and then one from the microkernel context back to the context to be used on resume. It is also not possible to partially save CPU state on an exception[2]. This is sadly unavoidable without hardware support for register banking. Please note that, apart from hardware and timer interrupts, switches between the user and the microkernel use the CHERI CCall instruction, which is not exception based, so context switches that result from syscalls do not incur any of this cost.

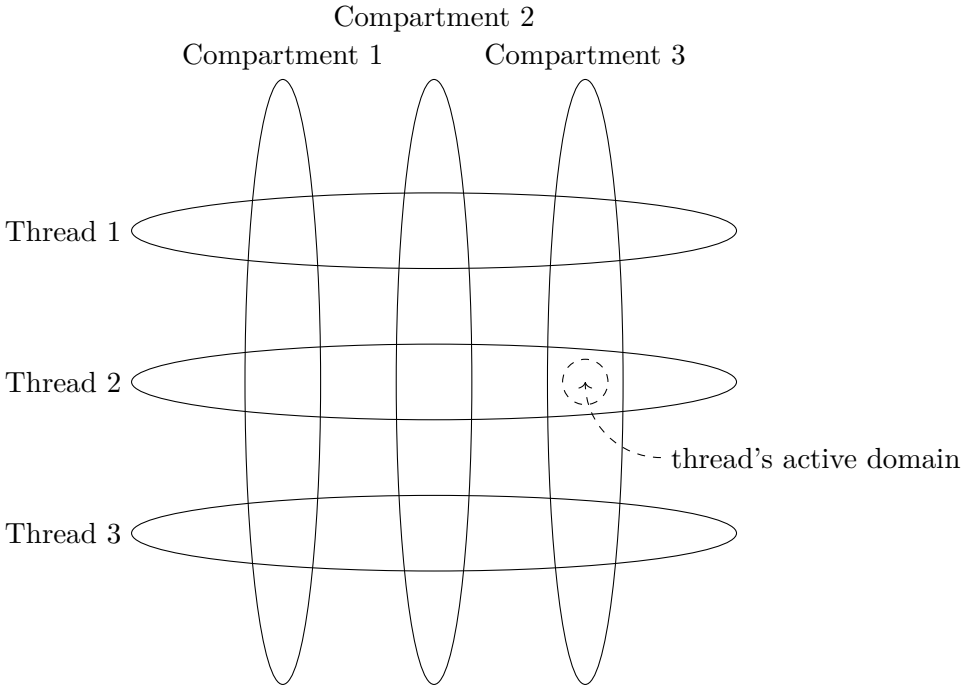


Figure 5.1: *Threads and compartments*

Domain local structure

Although communication can be achieved via serialised asynchronous message sends, CheriOS also supports a synchronous model that is predominately used to compartmentalise within processes. Threads and compartments are orthogonal, and as threads execute they may move synchronously across compartments. For each compartment a thread needs to run in, a thread has a domain (figure 5.1), which is unique to it. A structure called the DLS (domain local structure) identifies and provides information on a domain. Each entry in the structure is capability-sized, and different entries are required by different layers in the system. The nanokernel, CheriOS C ABI, and some specific objects all have their own entries. The entries required by the nanokernel are all used to provide user-handled synchronous exceptions, and are discussed in section 5.1. Entries for the CheriOS C ABI represent a thread's domain in a compartment. The layout of the DLS is shown in figure 5.2.

Every domain has a sealing capability called the DS (domain sealer). The DS is used to seal objects that should not be accessible outside of a particular domain. DS is also unavailable outside the domain. For example, function entry points will be sealed with the DS. Every object sealed with the DS must have the first field at offset 0 in the DLS. This first field is called the guard. The guard serves two roles: firstly, it is an atomic variable that guards access to the object, and, secondly, it acts as a demultiplexing key for objects with different types that have all been sealed with the same architectural type. Architectural sealing types are in short supply, so only one is provided to each domain by default. The guard allows code to distinguish between objects

[1]The nanokernel implements the same walk that hardware walkers conventionally perform.
[2]Unless handled by the user, who is allowed to see their own unsaved state.

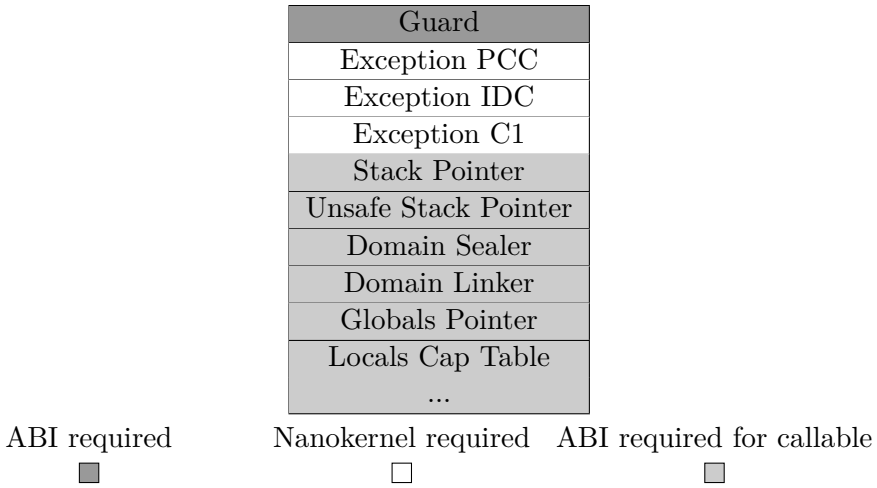


Figure 5.2: *Domain local structure*

of multiple semantic types, e.g., a C++ object, a return continuation, a callable function, a user object, etc.

The other fields are used to store ABI related capabilities. Briefly: SP (stack pointer) / USP (unsafe stack pointer) are used to load and store stacks when transitioning between domains. DS (domain sealer) is the capability used to seal objects from the current domain, as already mentioned. DL is the domain link function and is used to select trust policy on the callee side of cross-domain calls. Its use is properly explained later in this section. GP (Globals Pointer) is a pointer to the globals captable for the current process. Further entries in the DLS of callable objects are thread-local captable entries. A thread has a unique callable DLS for every domain it has to transition to. As the captable in the DLS is private, this ensures that threads cannot access another's thread-local variables.

The nanokernel requires that IDC (a capability register), unless null, at all times points to a DLS. The reason that we force this convention from such a low level of the system is due to the architectural significance of IDC. IDC is atomically installed alongside PCC when using CCall, the instruction used for domain transitions. The contents of PCC and IDC are therefore the only capabilities that can always be trusted, and the only ones whose values are atomically updated on domain transition. Everything a thread will need to execute in a compartment is loaded from its DLS, pointed to from the trusted IDC register.

CheriOS ABI

Calls between compartments are intended to transfer both control flow and data values, which may include capabilities. At a language level, programmers are explicit in the values they wish to pass and return, and also in where they wish control flow to go. However, at a machine level, much more is possible than at a language level. Without care, control flow can be transferred anywhere, and registers may happen to contain data a programmer did not wish to transfer. On a capability architecture, unintended data-flow is not only a concern as it entails an

information leak, but also because it leaks privilege. CheriOS compartments are only isolated as long as their capability graphs remain partitioned, and a single leaked capability can cause compartmentalisation to collapse.

The machine-level interface that defines how calls should be made is called an ABI (application binary interface). Mostly for performance reasons, most ABIs include unenforced agreements between callers and callees. Callers promise to only call specific entry points, callees agree to leave certain registers untouched. This requires callers and callees to be mutually trusting. It is the goal of CheriOS to allow mutual distrust between compartments, so when calls cross compartment boundaries, we need extra enforcement.

Calling convention

The CheriOS ABI includes the calling convention necessary to transition between different compartments with flexible trust relationships. The convention is secure enough to transition between the user and the kernel, is exception-free, and is implemented using only CHERI instructions, without requiring a third-party compartment. The convention restricts data-flow between compartments, at a machine level, to only intended values (explicit arguments or implicit arguments added by the compiler), and enforces a weak form of CFI (control-flow-integrity), which we call compartment-boundary CFI. Compartment-boundary CFI only enforces edges between compartments, and always by the compartment that is the destination of the edge, not taking the source into account. In other words, each compartment ensures (for itself) that returns are made in order and to the correct address, only valid function entries are called, and each thread is only running in a compartment at most once. This is not strong enough to guarantee global CFI, but is sufficient to protect a compartment from its own point of view.

The convention is configurable depending on how much each compartment trusts every other. The trust relationship between two compartments may be asymmetric, and we identify different types of calls as follows.

- (un)trusting - the caller (dis)trusts the callee
- (un)trusted - the callee (dis)trusts the caller

To aid remembering which is which, simply remember we always take the point of view of the caller. In the mutually distrusting case, we use untrusting/untrusted calls. On the other hand, a program that trusts a library might make trusting/untrusted calls. Callbacks from that library would be untrusting/trusted. The choice of whether a call is trusting is always made by the caller, and whether a call is trusted is always made by the callee. This allows programmers to make a performance/security trade-off, on a case-by-case basis if need be. By default, the distrusting modes are used across all dynamic-library boundaries, meaning that compartment boundaries are (by default) library boundaries. However, this is configurable, and compartments can be as small as single functions, or as large as entire programs.

Basic convention In order to maintain good performance, the basic calling convention assumes that both parties are trustworthy. Distrust is achieved using trampolines, on the caller's side to make untrusting calls, and on the callee's side to receive untrusted calls. Here we first describe the basic convention, which is what *should* happen if both parties behave well.

The callee promises the caller to:

- Save callee-save registers.
- Not read non-argument registers.
- Return to the correct address (with the correct return data), return in order, return only once, and return only when the thread is not already in the compartment.

The caller promises the callee to:

- Jump to a proper target address (with the correct target data), and jump only if a thread is not already in the domain.
- Not read non-explicit return registers on return from the call.

CheriOS represents machine-level invocable objects with a pair of capabilities. The first is a code capability, which points to code that should be executed. The second is a data capability, which is a capability to a per-thread DLS for the compartment that contains the code. When invoked, the code capability is installed in the program counter (PCC), and the data capability in the invoked data capability (IDC).

The CheriOS ABI is based on the CHERI-MIPS C ABI, albeit with a few departures. In the CHERI-MIPS C calling convention, as with most standard calling conventions, we can identify for a call a triple of code addresses: the target address (a pointer to the function entry), the return address, and the address stored in the program counter (PCC). On a call, the target address should in installed in the program counter, and a return address specified (often the program counter + 4). On return, the return address should be installed in the program counter. Each of these is assigned a register.

CheriOS adds another triple to the ABI to handle the data component: the target data, the return data, and IDC (the data equivalent to PCC). The rules for making calls and returns follows the exact same pattern as it did with code addresses. On a call, the target data should be installed in the IDC, and a return data specified (generally the same as the current IDC). On return, the return data should be installed in the IDC.

The action of simultaneously installing a value in PC and IDC can be achieved on CHERI-MIPS with a single CCall instruction, see section 2.4. The target data and target address can be either sealed or unsealed (as long as they are consistent), and so the same CCall instruction

can be used regardless of sealing. If the target data is the same as IDC (which means this is a call into the same compartment), a normal jump can be used instead of a CCall. Therefore, the convention decays back into the standard CHERI-MIPS convention for intra-compartment calls.

The CheriOS ABI is also slightly different from the CHERI-MIPS C ABI in a few other ways. Firstly, there are two stack pointers: the safe stack pointer and unsafe stack pointer; recall section 4.1. There are also two captables (the CHERI equivalent of GOTs).[37] One, the 'globals table', is shared between all threads in a process. The other, the 'locals table', is per-thread. In total, this requires an extra two registers (over standard CHERI-MIPS) always be in use, which results in extra register pressure. As CHERI-MIPS has 64 registers, this posed no real problem. In fact, some registers were not in use by the compiler at the time of implementation, and were spare for such a purpose. The in-development CHERI-RISC-V and Morello architectures will likely only have a more standard 32 registers; the effects of the extra pressure on such an architecture have not been measured.

Otherwise, all conventions regarding argument registers, return registers, caller/callee saves and temporary registers are consistent with standard CHERI-MIPS. All the new ABI registers are callee save, and assumed to be live-in to functions. However, all ABI related pointers can optionally be reloaded from a thread's DLS. This allows a thread to reload ABI-related state when only its IDC register is trusted.

Untrusting calls If the caller is untrusting, then it operates under the assumption that the callee is not keeping the promises identified in the basic convention, and so must enforce them itself.

First, the caller saves all callee-save registers to its stack. It then constructs a code/data pair that should be called in order to return. The code component will be a return-edge trampoline that reloads the callee-saved registers and performs CFI checks. The data component will be a capability to the stack where registers have been saved. The caller seals both these capabilities (using the domain sealer, which is not available outside the current domain) so that they cannot be modified or used by the callee.

Finally, just before making the call, all registers are cleared apart from: argument registers, the pair that represents the target function, and the pair that represent the return continuation. CHERI features an instruction to efficiently zero multiple registers for this purpose. Clearing registers ensures that if the callee does read non-argument registers it will only see zeros. By saving registers itself, the caller no longer has to trust the callee to do it. Finally, the return-edge trampoline enforces the CFI checks we require for compartment-boundary CFI. Returns cannot be made twice, as the variable that records whether or not a return has already been taken is stored in temporally-safe memory. If any check fails, a trap is trigged. This can either be handled by the compartment itself (see section 5.1), or will cause an unrecoverable error which will be propagated to the microkernel.

Untrusted calls As was possible for the caller, the callee can also enforce the caller's promises itself. The code/data pair for function entries are sealed by the callee before being released to the untrusted caller, this time with the domain sealer of the callee. On entry, the only data register which can be trusted is IDC, as it will have been installed using CCall. By design, every other value that needs to be loaded (such as a stack) can be reloaded from IDC. The callee does so, and also performs a CFI check to ensure a call is not being made while the thread seems to already be running in the compartment. On exit, the callee zeros registers that might leak unintended values before returning.

For both types of distrust, CFI checks are performed using a compare and swap on the guard variable of a thread's DLS. The guard variable records both whether the thread is currently in the compartment, and what the current call depth is. This allows compartments to ensure a thread is not running twice, and returns are made in order. Sealing the code component of an invocable object ensures only valid entry points are used, both on a call, and a return. See appendix B for more details and pseudocode for making calls.

Synchronous exceptions

A mechanism is required to deliver synchronous exceptions to users, such as result from traps or unaligned accesses. However, when the nanokernel forwards exceptions to the microkernel, the entire state of the CPU is sealed in order to protect user state. As such, the OS cannot modify user state, and so upon restarting a CPU context the same exception will inevitably occur again.

In order to facilitate user-handled exceptions, the nanokernel steps in. To reduce complexity in the nanokernel, only a very simple mechanism is offered. On the event of a 'user-handleable' synchronous exception, three registers are swapped with entries in the DLS: the program counter, the invoked data capability, and a scratch register, C1. Effectively, synchronous exceptions force a CCall, while also saving the previous PCC, IDC, and C1. If this fails, or IDC was null, user exceptions are propagated to the microkernel identically to asynchronous exceptions.

Because most state is not saved and sealed for user exceptions, it is crucial that exceptions be delivered to the correct compartment, and that this change atomically as we transition between compartments. By putting the exception-handling function pointer in the DLS, this is achieved automatically when using CCall. That is, even if a synchronous exception is taken immediately after a CCall, the callee's handling function will be invoked, not the caller's.

The nanokernel also offers three exception handling functions: `exception_getcause` to get the last user exception cause, `exception_replay` to replay the user exception if they cannot handle it, and `exception_return` to restore the compartment if the handler believes it fixed the cause of the exception. When return or replay are called, the values in the DLS are used to restore PCC, IDC and C1 (which the exception handler is allowed to change). The exception-related values in the DLS return to their original values so a new exception can be taken.

Exceptions cannot be nested; if another is taken before replay/return then the exception is propagated to the microkernel. Complicated exception-handling mechanisms such as saving

more registers, calling into a C function, or offering nested exceptions are left to the runtime to implement. The mechanism offered by the nanokernel is sufficient to implement schemes of arbitrary complexity, but simple enough to not add too much complexity to the nanokernel.

Because common classes of synchronous exceptions that were traditionally handled by the kernel, such as unaligned accesses, now need to be handled by user programs, a large amount of code risks being duplicated. See section 5.3 for how CheriOS's attestation primitive (a foundation) can be used to deduplicate this code in a mutually-distrusting way.

Fast leaf calls

A common practice in software engineering is to write small functions that do very little work, with the expectation they will be inlined by the compiler. With strong compartmentalisation this poses a problem. Simple functions that just get or set a variable now cannot be inlined across compartment boundaries. The cost for an untrusting call, just to execute as little as a single instruction, to then transition straight back, is prohibitive. In order to address this problem, the CheriOS ABI supports a second calling convention for leaf functions. This convention is as secure as untrusting/untrusted cross-compartment call, but with roughly the performance of a standard intra-compartment call, see 5.4.

Effectively, fast-leaf functions are run in the domain of the caller, but the caller verifies called code using attestation primitives at dynamic-link time. The attestation primitive (foundations; see section 5.3) allows verifying the contents of memory, and guaranteeing that these contents cannot change. To publish fast-leaf functions, the dynamic linker passes a sealed pair of capabilities as per usual to be used with CCall, but also passes a token allowing attestation of the code pointer. The callee code is not allowed to jump to register contents or handle exceptions, and is only allowed to use instructions that access argument registers (which can also be used as temporaries), or read from IDC. As jumping via a register and taking user exceptions is not allowed, callees cannot escape the bounds of the sealed code pointer, so the linker is able to verify all of the code potentially run.

It is trivial to check which instructions are contained in a memory region, and which registers they use. It is slightly more difficult to ensure that the called code is not able to handle user exceptions. Recall, to handle their own exceptions, callees would have to specify a handler at a specific offset to IDC. It is generally impossible to verify that the target DLS will not contain a capability at the required offset, especially as other writeable copies will have unconstrained use. Instead, we use capability bounds to our advantage and make sure that the nanokernel cannot read the exception entries, even if they exist. The fast-leaf ABI requires that the sealed DLS passed has a negative offset, large enough so the exception entries of the DLS would be out of bounds. Fast-leaf code cannot install a new value in the IDC register, so has no way of having in-bounds exception entries, so the nanokernel will always fail to read them if a user exception occurs. Again, it is trivial to check that IDC is never written to, and checking the offset of a sealed capability is also easy. Checking code contents requires some disassembly, so should probably be centralised and performed by a single checking service which can sign the result using more of the primitives found in section 5.3.

As a fast-leaf callee cannot read or write most registers, such as the stack, the caller does not need to protect them. Also, it is easy on the callee return side to know exactly which registers have been used and zero them. An example use of this convention in CheriOS is compartmentalisation of the TCP stack from the data being sent. All outgoing data pointers in the TCP stack are sealed so they cannot be read. This poses a problem, as the TCP stack needs to perform a checksum, so needs to make repeated calls into the compartment authorised to perform the checksum on packet fragments. Adding to a sum total is one of these fast-leaf functions. The TCP stack can verify the code it is calling, but still otherwise not trust the compartment that is allowed to see the data.

Reaching the nanokernel and microkernel

The cross-library calling convention is sufficiently protected so both the microkernel and nanokernel are reachable using it, and both kernels present as dynamic libraries. Capabilities for both nanokernel and microkernel functionality are installed as function pointers similarly to any other library, which allows easy virtualisation and interposition of syscalls. Furthermore, transitioning does not require a heavyweight exception mechanism.

This flexibility is, however, a security issue for nanokernel functionality. Users need to ensure capabilities they have for a particular nanokernel function are not a capability to a malicious man-in-the-middle function. For example, the nanokernel functions described in section 5.2 give guarantees of uniqueness for the memory capabilities they return. If it were possible for those calls to be interposed, then those semantics would be unenforceable.

For only the purpose of limiting interposition is the 'syscall' instruction used. The syscall instruction triggers an exception which the nanokernel handles. This is not used to invoke nanokernel functionality directly. Instead, the nanokernel syscall interface allows the user to request specific nanokernel capabilities by index. This allows programs running on CheriOS to 'de-virtualise' their interface to the nanokernel rather than just accepting whatever the dynamic linker provides.

This does not mean that every user can request any nanokernel capability, as most are too powerful to hold. Instead, the nanokernel provides bearer-token capabilities representing the authority to request a nanokernel capability via syscall. These are delegated instead of nanokernel capabilities themselves, allowing the OS to restrict which are available to users. If the OS falsifies these tokens, then requesting capabilities from the nanokernel will simply fail, and the user will not suffer any confusion.

Creating new domains

The distrusting ABI described assumes a DLS is already created for every compartment a thread may wish to enter. A C thread may enter any compartment in a process, and so every time a thread is spawned, a DLS is created in every compartment, although there is no reason why they could not be created dynamically on first entry. Because the DLS contains sensitive

capabilities, such as those used to seal it, it cannot be created externally to its compartment. For the purpose of this example, we will assume compartment boundaries always align with libraries.

Libraries will not release capabilities to internal global symbols, nor will they release the sealing capability to make new DLSs for a thread. This limits the role a dynamic linker can play. Instead, shared libraries contain a small sub-program called a 'link-server'. The *link-server* belongs to the same compartment as the library proper. The link-server does not require help from a dynamic linker to set itself up. In order to ensure both the link-server and library itself are not constructed with an additional trojan, a foundation is used; see 5.3.

To create the first, or subsequent, threads for a process, messages are sent to the link-servers of all the dynamic libraries the program needs. These messages are sent using the microkernel's IPC mechanism. Although IPC is untrusted, the capabilities sent over it can be protected from adversaries using the protection primitives offered by foundations. Senders of capabilities can lock them such that only the correct link-server can unlock them. Furthermore, link-servers can sign the capabilities they return. Assume that every message sent during the link process is signed by the sending link-server, or locked for the recipient link-server, as appropriate.

The first stage in linking is sending messages to each link-server requesting how much memory is needed for new DLS structures, and other assorted data structures such as stacks. The application then allocates reservations for these structures and sends them back to the link-servers. A link-server will use the reservations to create a new DLS structure, will seal a capability to it, and return it. The second stage exchanges symbols with external linkage, including the DLS structures just created as they constitute the target-data component of cross-library functions. Each library will directly communicate with others to request symbols by name. Libraries will refuse to release symbols with internal linkage, and may only be willing to expose symbols to trusted libraries. Foundation mechanisms are used to enforce this. Symbols are then used to populate the capability tables for both local and global variables.

Finally, now DLSs have been created and populated for each compartment, calls can be made. A call to the initialisation function of each library is made. This initialisation function is not part of the link-server, but the library proper. The result of this somewhat complicated process is that libraries can enforce exactly which symbols are visible to other libraries. Neither the microkernel, nor program-loading services, ever get copes of capabilities that are shared. Thread-local storage is truly thread local, unless otherwise shared. Link-servers do see every capability used to set up corresponding DLS structures, but they are small and compiler-generated. Link serves do not keep copies of the capabilities they receive when new threads are created.

Although otherwise normal dynamic libraries, the nanokernel and microkernel are special cased. The nanokernel only has a single DLS for the entire software system, and as many threads as desired can use it at once. The microkernel creates a DLS for each object activation, so that any object activation can be active in the microkernel simultaneously. Neither the nanokernel nor microkernel are willing to share symbols with programs, so the symbol exchange step is vastly simplified. The nanokernel releases capabilities to functions using syscall, and the microkernel passes a table of capabilities when it starts up new activations.

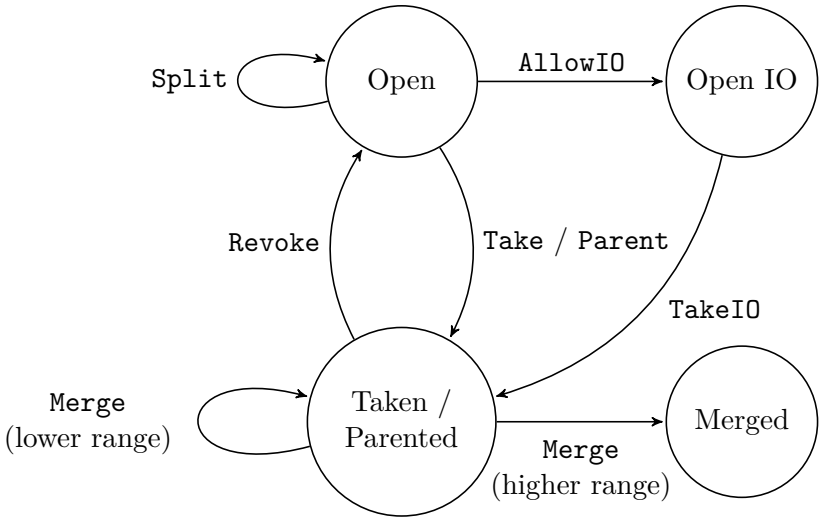


Figure 5.3: *Reservation state diagram*

5.2 Memory management

One of the roles of the OS is to manage memory resources. The OS requires that users be unable to arbitrarily access memory, and requires the ability to allocate, track, and potentially revoke access. The user, on the other hand, requires that the memory they do access is private (unless they explicitly share it), has a lifetime they can reason about, and has understandable semantics. CHERI capabilities are derived monotonically, which means a user can be assured that for any memory capability they receive from the OS, the OS will have a copy. At first glance, this would imply that either the user cannot have capabilities the OS does not, or the OS will have to give up control of memory, neither of which we can allow. The key to resolving this tension between requirements is an introduction of a new primitive, called a Reservation, that de-conflates the concepts of right-to-access, and right-to-allocate.

Reservations

Overview

Reservations are a primitive offered by the nanokernel that allow delegating the authority to allocate memory, without possessing the memory capability itself. This is in contrast to capability OSs that have memory capabilities representing the authority to map or otherwise access memory pages. Reservations also give guarantees on the semantics of the memory covered by them, ensuring the memory behaves roughly like physical memory would. A memory reservation is an opaque handle that confers the right to allocate a specific virtual-memory range. Multiple copies of a handle can be made, but operations on reservations are atomic and apply to all copies. A nanokernel capability exists to get a reservation for all of virtual memory. This capability can be exercised only once, and is normally invoked during boot.

A reservation is in one of fives states: 'Open', 'Open IO', 'Taken', 'Parented', or 'Merged', see figure 5.3. An Open reservation represents an unallocated memory range. Open reservations can be `Split` to a particular length. Splitting a Reservation sets its (and any other copies of the handle's) length to the specified amount, and returns a new Reservation handle for the remaining space. Splitting is used throughout the system to divide the memory resource. The OS's memory manager splits reservations in multiples of pages. Programs' malloc implementations split these ranges into individual objects.

The `Take` function allocates the range described by a Reservation. Upon a successful `Take` operation, a memory capability to the range is returned[3]. The nanokernel makes several guarantees about the returned memory capability:

- The capability is unique and does not alias or overlap with any other memory capability.
- No further capabilities that alias or overlap with the region will be granted by the nanokernel, unless every capability to the region has been revoked.
- The contents of the memory pointed to by the capability are entirely zeros.

Assuming a process can protect its copy of a memory capability from leaking, successfully taking a reservation returns memory with both integrity and confidentially for the lifetime of the system, even if the entity that provided the reservation is untrusted. This mechanism is used extensively in CheriOS, not only to allocate memory for the user, but for the user to allocate for the OS. Taking a reservation transitions it into the Taken state. Taken reservations cannot be taken again and cannot be split further. They do, however, still have use. A taken Reservation, combined with another nanokernel capability, grants the authority to revoke that range. Revocation is described in section 4.2, and transitions a Taken reservation into Open again. However, rather than needing to maintain multiple Reservation handles and individually revoke them, Taken Reservations can be merged. The `Merge` function merges two adjacent reservations into one. The handle for the span with the lesser address then represents the entire range, and is still Taken. The other Reservation is marked as 'Merged', which means the handle is unusable.

Split is destructive, making it difficult for the OS to keep a handle on its memory for revocation. If a reservation is split to a length of zero, other holders of a handle also have a reservation of length zero. To solve this problem, a `Parent` function is offered. Only an Open reservation can be made a Parent. Doing so creates a new open reservation for the range, and sets the old Reservation into the parent state. Parent reservations are effectively the same as Taken reservations. No amount of splitting of a child Reservation will affect the parent. However, only reservations with the same parent can be merged. Parents are used whenever memory that will need revoking is delegated.

Although the structure is never traversed, notionally, reservations describe an n-ary tree, where walking the leaves would result in walking contiguous spans of memory across all memory.

[3]In fact two are returned: one writable, one executable. This is due to enforcing W^X for all capabilities in the system.

`Split` gives a leaf a new sibling node. `Merge` merges two siblings into a single node. `Parent` turns a leaf node into an internal node with a single child. `Revoke` takes any node in the tree, destroy it and any children, and replace it with a fresh one.

The remaining state is 'OpenIO'. Open IO reservations are made from open reservations, and are a declaration that any part of the range may map to IO space. These reservations can only be taken, and a different `TakeIO` call is used so that callers are aware the resulting memory capability may not have RAM semantics. The returned capability is unable to load or store capabilities. Constraints are relaxed in that after taking the reservation the 'contents', or rather memory mapped registers, may be non-zero. Constraints on uniqueness are maintained. An IO reservation may still map to RAM for any of its pages. In the case it does, the all-zeros and non-aliasing guarantees are not relaxed. Consider the following example for hot-plugging a new device: the device is plugged in, and a message is sent to the device manager about a new device at a specific IO region. The device manager allocates a new open reservation large enough to map to the IO region. It converts the Open reservation into an OpenIO reservation, and asks the memory manager to map it to the physical IO region. Then, it securely loads a driver to manage the device and passes it the reservation. The driver calls `TakeIO` on the reservation, and the result is a memory capability that uniquely maps to the IO region. If the driver crashes, the device manager still has a copy of the reservation to call revoke on. The device manager never acquires the memory capability itself, so cannot manipulate devices concurrently with drivers.

Implementation

Reservation handles are sealed capabilities to a metadata node in memory, only unsealable by the nanokernel. Crucially, metadata is stored in-band (in the virtual view of memory), directly before the range it describes. Doing this has many advantages needed to make reservations practical.

- Merged metadata can be unmapped along with the ranges they describe - if memory is finished with, the node is merged, and its backing is freed. This allows the amount of memory needed to track reservations to stay constant as physical memory is reused.
- Merged handles are revoked concurrently by the same mechanism that revokes memory - When a reservation is revoked, metadata handles also need revoking. Revoke sweeps are quite expensive, so hopefully the region under revocation is result of merging thousands to millions of metadata nodes. If they were stored out of band, a single filter register would be insufficient. By storing them in band we get free collection of the handles.
- The nanokernel needs no allocator - allocating new metadata nodes is now trivial. The nanokernel can be assured that the nodes are safe to use as reserved regions will not be accessible outside the nanokernel.

Reservation metadata nodes are 16 bytes, the same size as a capability. Since reservations are to be used as for every allocation, metadata nodes would create an unacceptable overhead for

Field:	State	Low Term	High Term	Scale	Length	Bitmap
Bits:	2	1	1	7	53	64

Figure 5.4: *Reservation metadata layout*

small objects. One byte objects would only achieve ∼6% memory utilisation. To circumvent this, a special `splitslab` operation can be applied to reservations. A `splitslab` operation is like a normal split, but the reservation is split into many equally sized ranges, which it also becomes the parent of. These slab-reservations contain no metadata node (the metadata is stored in the parent) but cannot be further split or merged. In the current implementation, a reservation can be split in up to 64 pieces, achieving a 98% memory utilisation for one byte objects. The supported object sizes grow geometrically. Every 4^{th} size doubles. These are the same slab sizes used by high performance allocators, e.g. SuperMalloc[74] and snmalloc.[80] Even with internal fragmentation, at least ∼88% of memory can be utilised with these bucket sizes. The allocator described in section 4.1 uses slab-reservations for its slab allocator.

Although most metadata is stored in the parent, reservations for individual objects in the slab must be distinguishable. This is done by storing the index in the reference. CHERI can store a pair of a capability and a small integer by abusing the capability offset field.[4] If a capability to an object has an offset of 0 (i.e. its bounds start where the object starts), then the offset field can be used to encode arbitrary information. In order to access the object, set the offset back to 0. The resulting pair can still be sealed, and neither can be changed without unsealing. This trick is similar to cramming bits in the bottom or top of pointers on traditional machines, but allows more bits to play with. If a slab reservation has an offset of 0, it refers to parent node. Otherwise it represents just one of the child reservations, and 1 is subtracted from the offset to give the index.

The layout of reservation metadata nodes is given in figure 5.4. The range's base is implicit in the node's location, but both length and state are stored. The length field is 53 bits, with 4 implied zeros due to alignment, allowing a large 57 bit virtual space. Two bits are used to enumerate the reservation state. The parent state is equivalent to the taken state, so only 4 states are required rather than 5. A single bit marks whether or not a reservation is the first or last of its siblings, ensuring children with different parents are never merged. The scale field gives the bucket size if `splitslab` is used. If the value is 0, it is not a slab. Otherwise, subtracting 1 gives an unsigned floating point number with a 2-bit mantissa and 5-bit exponent. If the exponent is 0 the number is denormal, otherwise the exponent is biased by 1 and the mantissa has an implicit 1 bit. This allows all integers up to 8 to be representable. The largest object is $7 * 2^{2^5-2} = 7G$, much larger than would practically be needed for a slab. Calculating the slab size from the scale requires only a few integer operations. The bitmap field stores whether or not individual slab objects are taken. Operations on reservations are trivial to implement; the nanokernel unseals the handle and performs an atomic CAS, generally only on the first 64 bits. As metadata is

[4] This trick is used extensively in CheriOS to seal pointer / integer pairs, when allocating memory for the pair would be prohibitive. For example, it is also used to generate reply capabilities for kernel message send, and the integer there represents a sequence number.

stored in memory, other references to the same range are effectively updated when splits and takes are performed.

Knowing how reservations are represented, it becomes apparent why certain restrictions exist. We only allow splitting open reservations and merging taken ones as this ensures metatdata will never be accessible outside the nanokernel. If we were to split a taken reservation, this would create a metadata node in the middle of an in-use region. If we merged an open one, then a metadata node would exist in memory that may later be taken. Even merged reservations, which can have no operations applied to them, cannot have metadata in memory used by the user. The handle will still be a valid capability, and metadata could be overwritten so the reservation was open and of arbitrary length. There is similar logic behind only allowing merging of reservations with the same parent. If a child were merged such that it was larger than its parent, a revocation of the parent may only partially revoke the child.

There are some downsides to the in-band metadata. Operations like split and parent steal 16 bytes from the range. This has to be accounted for when writing allocators. Furthermore, alignment is often sacrificed. For example, the CheriOS page allocator returns lengths shorter and a base slightly greater than page alignment. If exactly one page is requested by a user, they will therefore get 2. However, most users only call `malloc` and then `take`, and so are not aware of this. Malloc on CheriOS knows to bias memory requests for the page allocator such that there is no wastage.

Some care also has to be taken using reservations around memory-mapped IO regions. The nanokernel puts metadata in-band, and cannot do this in IO pages. The solution is to have the reservation metadata node in a different page than the IO window. This first page can be mapped to RAM (to contain a metadata node), and the rest to an IO window. A whole page is not necessarily wasted memory; the remaining space can be used for other purposes. In-band metadata is why IO reservations are allowed to map to both RAM and IO pages. To create an open reservation, at least the first page will already have been mapped for the metadata node. Once converted to an OpenIO reservation, no more splitting is allowed as any page (but the first) could then be an IO page, so cannot store metadata. The nanokernel disallows open reservations from mapping to IO pages. There is no way to get back from an OpenIO reservation to a normal open reservation without going through revocation. Revoking will unmap pages, maintaining the property that metadata only ever get put in RAM.

Type reservations

There are also reservations over sealing capabilities, so called 'Type Reservations'. In principle, they function similarly to memory reservations, allowing one-time allocation capabilities to seal/unseal with a particular type, and support a `take` and `revoke` operation. However, they are not range based. To get a type reservation handle for any type, it suffices to have a single nanokernel capability to request type reservations. This capability is given to a Type Manager, which then tracks owners of types (Type Ownership Principals, or TOPs), granting them type reservations, which they in turn take to obtain unique sealing capabilities. The same hardware-assisted sweep that revokes a range of memory can revoke a range of types. A single bit per type

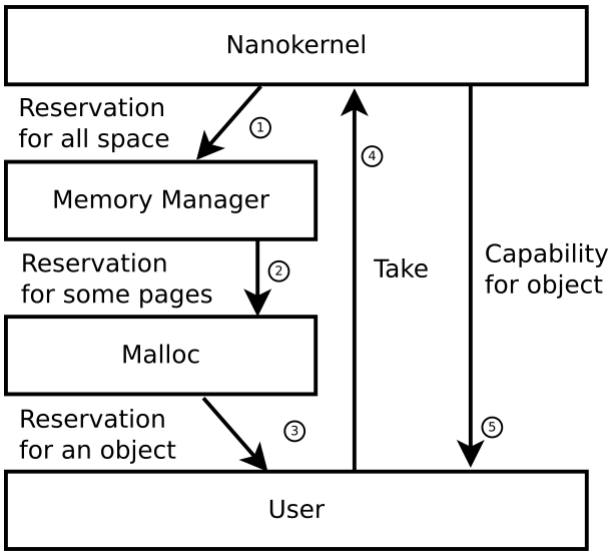


Figure 5.5: *Using reservations to allocate objects for the user*

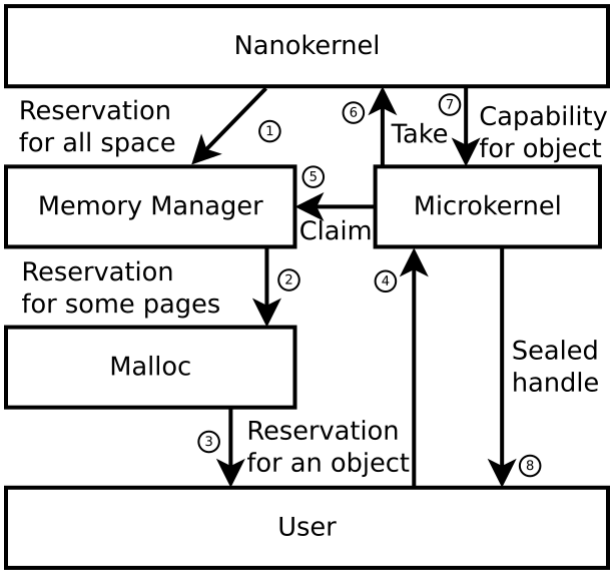


Figure 5.6: *Using reservations to allocate objects for the kernel*

is needed, and there are not that many types, so metadata detailing which are taken is stored out of band statically allocated.

Example uses

Here I will illustrate two example usages of Reservations to allocate objects, one for the user (figure 5.5), and one for the microkernel (figure 5.6), where both distrust the other. The story for both allocations begins similarly. During OS start-up, the memory manager exercises its capability to get a reservation for all virtual space (1). It splits this reservation into chunks of

pages, passing one chunk (2) to a user's malloc implementation to use. Malloc further sub-divides this reservation into individual objects, and passes one to the application (3).

In our first example, the user calls `take` (4), and a memory capability is returned (5). This flow of reservations keeps the Memory manager and malloc out of the TCB for the user. A bug or malware attack by either the memory manager or malloc could still result in a reservation being revoked too early, or not provided at all. Early revocation results in a clean exception and the program is stopped. At no point can malloc or the memory manager get access to the contents of memory that the user can.

The second example differs in that it is the microkernel that requires memory. Recall in section 3.4 that the microkernel does not perform memory management, but rather it is made the responsibility of the user. Rather than ask the memory manager for memory directly, the microkernel requires that users provide reservations to syscalls (4) that would require allocating a new object. For example, this is used to create new object activations. If needed, the microkernel claims (5) the virtual space (see section 4.3) to ensure it is not unmapped. At step (6), the microkernel calls `take` to get a memory capability (7), which it seals and returns to the user (8) as a new kernel object handle. This method of allocating kernel objects has some benefits: it puts the onus of finding memory onto users, which in turn counts it towards their resource limits. OSs such as Eros[118] use this allocation approach pervasively, although via a different mechanism than reservations. Failing to attribute kernel metadata to a user's allocation limit can lead to denial-of-service attacks. Having users allocate kernel structures for a process also results in structures being co-located on that processes heap, leading to better spatial locality.

Guarded management

Most programs, including all that use reservations, use virtual memory rather than physical. Amongst its other benefits, virtual memory allows us to over-commit when allocating contiguous space for new programs, and offers a significantly larger addressable space than can be physically backed. This is instrumental in delaying how often we must revoke reservations, see section 4.2. However, misuse of memory mapping capabilities by the OS could thwart our other attempts at providing security. The capability to remap virtual pages to arbitrary physical pages, combined with a capability to any virtual page, is functionally equivalent to the capability to access all of physical memory. Obviously, we cannot allow the OS such unrestricted access. At the same time, the nanokernel aims at enforcing minimal policy, leaving decisions with regard to layout and lifetime to the OS.

Tracking physical pages

The nanokernel contains a structure that summarises the state of physical memory. This structure is an indexable array of records, where each record summarises a range of pages. The nanokernel releases a read-only capability to this table, so all logic that only requires reading the table is outside the nanokernel. Each page is in one of 10 states, see figure 5.7. When the nanokernel initialiser has finished running, pages used internally for statically-sized structures

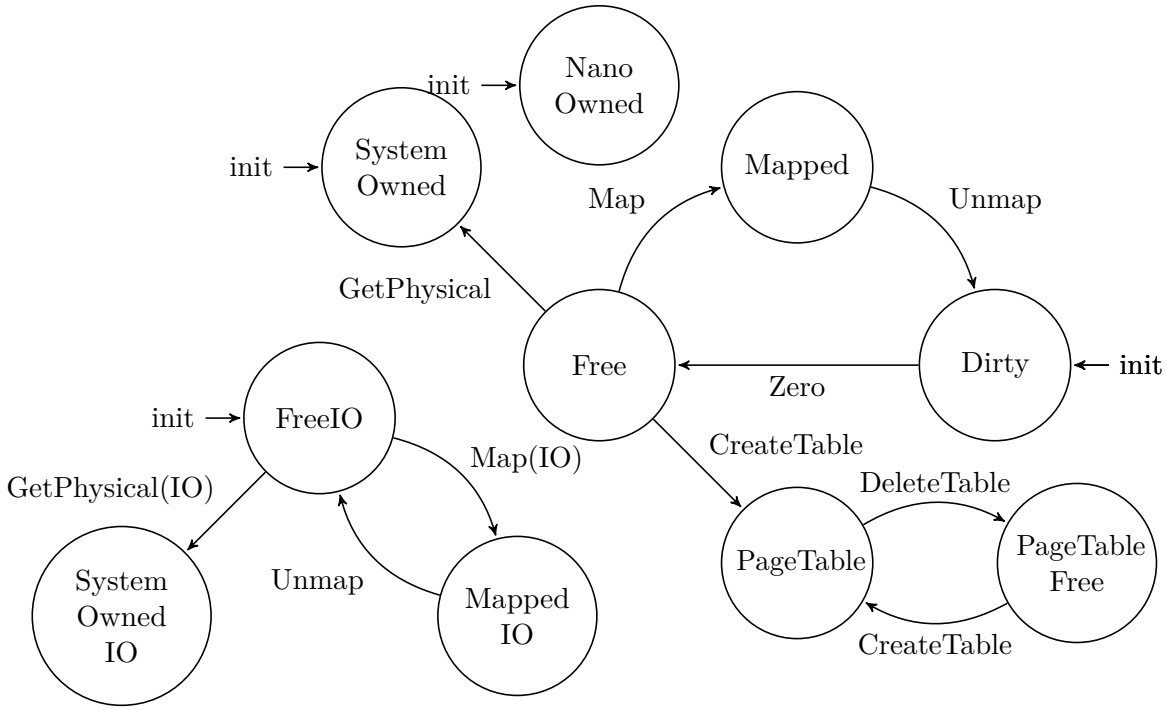


Figure 5.7: *Physical page state diagram*

(such as the physical record itself) are marked NanoOwned, pages containing the microkernel image are marked SystemOwned, pages that are IO windows as FreeIO, and the rest Dirty. Pages start as dirty so data and capabilities are not leaked across a reboot. Before any physical page can be used, a nanokernel call to zero it must be made first. This ensures no data leaks between subsequent uses of a physical page.

Many nanokernel functions require the caller to specify an index into the physical pagetable. For example, creating a new pagetable requires one physical page. The caller (i.e. the OS) must ensure the record at the index be of the correct length. The nanokernel provides functions to split and merge ranges in the table. This allows most of the logic for managing the physical pagetable to be pushed out into the OS. Only the writes to the record itself are inside the nanokernel.

The OS can request that 'Physical', i.e. directly-mapped capabilities, be released. Doing so marks physical pages as SystemOwned, and will no longer be usable to back virtual memory. This is intended for limited use where virtual memory is not desired. The CheriOS microkernel *code* lives in SystemOwned directly-mapped memory, but this is not a necessity. In systems that do not support 'Physical' addressing, and have a hardware pagetable walker, providing physical windows means having pagetables with trivial translations. These could set these up at boot.

Physical pages can be used for pagetables. When freed, these pages are only then reusable as pagetables. This is an implementation artefact. The OS can currently request read-only, direct-mapped views of the pagetables for convenience. These are harder to revoke than virtual memory, so the nanokernel requires type stability for pages dedicated to holding pagetables.

Lastly, physical pages can back virtual ones. The nanokernel requires that the page not already be in use and be already zeroed. This maintains the guarantees on Reservations detailed in section 5.2. When unmapped, the physical page is marked as dirty, and will need zeroing before becoming eligible for use.

The nanokernel differentiates between IO pages and RAM pages. Most states have an IO counterpart, and similar operations can be performed on them. The only difference with IO pages is that the returned capability (either via requesting physical capabilities or by using reservations) cannot store capabilities, and the nanokernel promises to not sweep them when revoking, as this may cause unintended side effects. To map IO pages, an OpenIO reservation is required to prove to the nanokernel that IO pages are not being used in place of RAM. When unmapped, IO pages are not zeroed. As they cannot be used in conjunction with standard reservations, this does not have the unintended consequence of breaking confidentiality. Users of IO reservations are aware that the virtual capabilities returned, if mapped to IO regions, may be used again in the future without zeroing.

Tracking virtual pages

The OS is not allowed to install arbitrary entries in pagetables. If it could, then any physical memory could effectively be accessed using a capability for only a single virtual page. In order to ensure safe management of memory, when a new mapping is made, the nanokernel insists that the virtual entry not have been mapped before, and the physical page be in the Free state (which will guarantee it being zeroed). This requirement is at odds with some contemporary uses of MMUs, where multiple mappings to the same physical page provide different permissions, or allow sharing pages between address spaces. This matters not in CheriOS; only one address space is in use, and capabilities restrict permissions. Even so, if remapping a virtual page or mapping to an already in use physical page were desperately required then it would be trivial to modify the nanokernel to allow it in a safe way. A caller would have to prove to the nanokernel that they already had a capability for the virtual/physical page, and so were not elevating privilege. This is not implemented in current versions as there was no need for it.

CHERI-MIPS, like MIPS, has a software defined page-table walker. In CheriOS, we put the walker in the nanokernel, although it performs the same walk hardware-based walkers typically do. Shootdown of the TLB is also handled by the nanokernel, to ensure stale mappings do not break invariants of multiple virtual pages not being allowed to map to the same physical page. This is not done every time a mapping is created, it suffices to shootdown the TLB when a physical page is zeroed only if a virtual page has been unmapped since the last zeroing. This allows multiple physical pages to be zeroed, or multiple unmappings to be performed, with only one shootdown.

The nanokernel maintains a conventional multi-level pagetable. Each entry is in one of three states, see figure 5.8. When the OS starts, all virtual pages begin in the unmapped state, described by a single top-level table. A nanokernel capability exists to get a sealed capability to this top-level table. Given a capability to a table, the OS can request a new sub-table be created, or a page mapped, or unmapped. The only requirement the nanokernel imposes when mapping

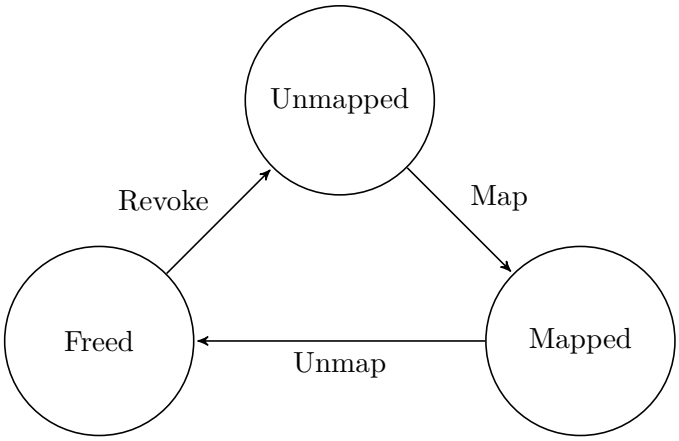


Figure 5.8: *Virtual page state diagram*

an entry is that it be in the unmapped state. When we unmap a page, the virtual page is not marked as unmapped, but freed. The physical page is immediately available for reuse; however, the virtual page is not. Even if a page is unmapped, capabilities may remain referring to the page. In order to maintain the guarantee of uniqueness offered by the nanokernel when taking reservations, the virtual page is only marked as unmapped after revocation; see section 4.2 for details of revocation. There is no need to maintain a page full of freed entries. Instead, the entry in the table one level higher is marked as freed and the pagetable reused. This allows CheriOS to, over time, consume a much larger virtual space than it has physical backing for, delaying the need to revoke virtual space.

Currently CheriOS does not support swapping, although this is intended future work. Swapping requires reconstructing capabilities from untagged memory, and also encryption of page contents so as not to leak information. Both of these can only be done by the nanokernel, as no other part of the system has either the authority or the trust. It is intended that a fourth 'swapped' state would be added to virtual page tracking (which would include a nonce to preclude replay attacks), and two new nanokernel calls would be added. The first would swap out a page. The OS would select a page to be paged out, and provide a buffer for the nanokernel to copy encrypted page contents into. The virtual page would be marked as 'swapped' in its PTE, and the physical page as 'dirty'. This encrypted block would contain both the data the page had (including tags), which virtual page the data belonged to, and the nonce contained in the PTE. The responsibility for saving this buffer to disc can be left to the OS. The second call would swap a page back in, and the OS should provide the virtual page to swap back in, the buffer with encrypted data, and a clean physical page to use. Swapped-out pages would also need sweeping during revocation, and it would be left to the OS to page them back in, in response to exceptions generated by the nanokernel. The CHERI architecture supports a number of instructions to efficiently rebuild capabilities for the purpose of paging[131].

5.3 Attestation

When used properly, other primitives in CheriOS, such as reservations, allow for fine-grained isolation. However, it is still possible to fool a program into thinking it is running correctly by simply changing its instructions. In truth, nothing can be done to prevent this. Any checks meant to verify the currently running program has been loaded correctly can be removed from a program. However, given we trust our own state, it is possible to attest that *other* programs were loaded correctly. This is, in fact, equally desirable. Programs will rarely start with sensitive data, or if they did, the OS would already have access. We are instead interested in attesting that the programs we release sensitive data or capabilities *to* were started in a good state, or at least as good as ours.

The CheriOS primitive that allows for attesting that programs began in a correct and well-understood state is called a foundation. Foundations, their features, implementation, and uses are covered in this section.

Foundations

A *foundation* is an abstract collection of program states, founded on a much simpler starting state, with the authority to perform 'foundation operations'. These operations allow foundations to either attach a starting-state-based signature to capabilities, or to unlock capabilities only intended for a specific foundation.

Previous work, such as Intel's SGX, allows a hardware-enforced compartment, or 'enclave' to occupy a single segment. Drawn as a capability graph, we might claim that compartments outside the enclave had a capability to all the space apart from the enclave's segment, and the enclave itself a capability to all space. In CHERI, we enjoy the flexibility of arbitrarily-complex graphs, and compartments that span disjoint regions of memory. This allows for more intricate set-ups, such as nested compartments, or compartments that share memory with different sets of permissions. Sadly, the complexity of such capability graphs makes it difficult to reason about how accessible a compartment is. To know if a compartment could be accessed, we would need to know exactly what capabilities exist, how reachable they are, and if they refer to the compartment. Measuring this for every compartment would be far too complicated to be practical.

Rather than impose restrictions on shapes capability graphs may eventually take, we instead only limit starting states, giving the name 'Foundation'. A foundation starts with a pre-initialised segment of memory with a set of entry points. Upon its creation, the capability graph for this segment is severely limited: the nanokernel guarantees that it neither contains any capabilities, nor will there be any that refer to it (outside of private nanokernel capabilities). At this point they are very similar to enclaves. After creation, capability graphs are allowed to grow, foundations can grant their authority to other pieces of software, but can still be identified by this initial segment.

A foundation does not necessarily correspond to any another CheriOS primitive. It is not bound to any CPU context, object activation, process, thread, or compartment. Any number of threads can be acting as a foundation, and dually, any thread may act with the authority of multiple foundations. However, in practice, we will often load each program compartment as individual foundations.

Creating a foundation

In order to create a foundation, a nanokernel function must be called. This function must be provided with a reservation to use as the foundation's initial segment, and a pointer to memory that contains an initialisation image. Furthermore, the caller must specify how many entry points the foundation is allowed to have, and an offset into the initial segment for the foundation's first entry.

The nanokernel will take the reservation so that no other part of the system can, and will copy the initialisation image into the reservation. As it does this, it will also calculate a SHA256 hash of the data it is copying. This hash, along with the image length, number of entries, and first entry offset form a *foundation ID*. The length is not strictly necessary to include in the hash as SHA256 takes it into account, but is useful as part of the definition of a foundation to users.

Foundations are identified by their foundation IDs, which are implemented as read only capabilities. It should be noted that the *address* of a foundation ID identifies the foundation uniquely. If the same foundation is loaded twice, the contents of the ID will be the same, but they will have different addresses. When attempting to attest a foundation, programs use the contents of an ID to reason about what was loaded, but then use the pointer itself when performing stateful interaction with the foundation. The same systems that provide temporal safety ensure that the same foundation loaded in the same location across the lifetime of the system will not be confused.

Once the nanokernel has finished the copy, and has created a foundation ID for the new foundation, it will also construct and return an entry token. An *entry token* is a sealed capability that allows the holder to simultaneously grant themselves the authority of the foundation and transfer into it. The location this first entry token will transfer to was specified as an offset into the initial segment when the foundation created.

When the copy is performed, any tag bits found are stripped. This ensures the first requirement on the starting state of a foundation, that the foundation starts with no capabilities to outside of itself. Also, because a reservation was used, the nanokernel can assure the second requirement, that no capabilities are available outside the nanokernel that refer to the foundation's initial segment.

Entering and exiting foundations

Initially, the only way a foundation can be entered is using an entry token with the nanokernel's 'foundation enter' function. An application can also request the foundation ID for a

given entry token in order to verify what code it will be jumping into. When jumping into a foundation, argument registers are preserved. This is initially how capabilities will be introduced to a foundation's initial segment. Furthermore, on entering a foundation, not only is PCC installed with the capability for the foundation's initial segment, but some extra arguments are provided. The first is a capability to read and write the foundation's initial segment. The second is an optional extra to make entry analogous to CCall, and is installed in IDC. Its use is described in more detail in section 5.3. The third, and most important, is a special *authority token*, which is a bearer token that proves the holder is acting with the authority of the foundation.

An authority token is required for all nanokernel functions that require the authority of a foundation. Rather than being an ambient property of the running CPU context, the state of being 'in' a foundation is being in any state that can follow its capability graph and find an authority token. Therefore, a thread can be considered to be acting as multiple foundations, and a thread need not have its program counter in a foundation's initial segment to be acting as the foundation. It should be clear at this point that foundations can outgrow their initial segment, or can discard it entirely. Some foundations will self extract, and others will grant their authority token to other pieces of code running in the system.

Exiting a foundation is, therefore, not a nanokernel function. Instead, access to the authority token must be sealed away, or discarded. This is possible using the same calling convention already in use for normal function calls. The CheriOS ABI already protects all capabilities that are not explicit arguments from leaking from a compartment. If a compartment wishes to make a call, and not implicitly grant the callee the authority of the foundation, it suffices to make an untrusting call. Return transitions need not use the nanokernel's 'foundation enter' function, they can instead return using the normal calling conventions which can unseal access to the authority token. The nanokernel-facilitated entry is normally only used for initialisation, and to initially grant the authority token. Because of this, apart from a set-up cost, having entire programs, libraries, or even functions in foundations confers no extra cost beyond the untrusting version of the ABI.

Passing data between foundations

Data can be passed into a foundation by multiple means. We could directly use an entry token, CCall a function that unseals access to the authority token, or place capabilities in shared memory. However, sometimes access to the foundation sits behind multiple levels of services and abstractions. The microkernel offers IPC, but its message send function should be viewed as an insecure channel. In order to allow secure communication across insecure channels, the nanokernel provides a set of encryption-like functions that allow for the concealment of capabilities.

Authority tokens are paired one to one with foundation IDs. An authority token can be used as either a symmetric key, or a private key where its corresponding foundation ID is the public key. These keys can be used to seal access to capabilities. This sealing is provided by the nanokernel and should not be confused with architectural sealing. Also, no actual encryption is used, but as the primitives are used to communicate by software components in similar ways as encryption is used in conventional protocols, familiar terms will be used.

A holder of an authority token can 'sign' a capability. A signed capability can then be 'verified' by any unprivileged user. Verification is a nanokernel function, and the nanokernel will not only provide the capability that was signed, but will also return the foundation ID that corresponds to the authority token used to sign the capability in the first place. This mechanism can be used by a foundation to prove to recipients that a holder of a particular authority token trusts a particular capability. For example, this mechanism can be used to publish a buffer where events will be stored, to prove that the events in the buffer have a particular foundation as a source. It is also used to sign callback functions, see section 6.1 for a use in CheriOS's IO layer. When creating a signed capability, the signer can request they only be checkable once. This makes them reservation-like, and is useful if the consumer requires they uniquely share the capability with the publisher.

A holder of a foundation ID can 'asymmetric lock' a capability. An asymmetric locked capability can only be unlocked by the corresponding authority token. This exists to allow users to pass sensitive capabilities using untrusted message passing channels, to only be unlockable by a specific foundation. For example, the holder of an AES key may lock it using the foundation ID for an AES routine they trust, and only release the locked key to the filesystem. This example is covered in more detail in section 6.3.

A holder of an authority token can also 'symmetric lock' a capability. A symmetrically locked capability can only be unlocked by a holder of the same authority token. This mechanism is useful when a foundation wishes to pass a sensitive capability to an untrusted party to be used as a callback argument. A special case of a symmetric locked capability can be constructed to make the capability 'Invocable'. An invocable symmetric locked capability can be passed when entering a foundation via the nanokernel, and will automatically be unlocked and installed in IDC. This allows foundation entry to behave like a CCall. Recall that what gets installed in IDC has special meaning for exceptions, see section 5.1. This automatic installation of IDC allows foundation to handle their own exceptions immediately on entry.

While data is kept in memory, these mechanism are sufficient to establish secure channels between entities in the system. It is preferential to encryption due to the high cost and complexity of encrypting data securely. Sometimes, however, data needs to be stored outside of memory. Memory on external devices is not covered by CHERI capabilities, so at this point we still need to resort to encryption. Remote attestation steps and facilitating translation between the local protection primitives and remote ones is currently out of scope of the nanokernel. In order to allow users to construct their own schemes, the nanokernel can currently provision 128-bit keys. These keys are derived from a secret nanokernel master key, and a foundation ID. Only the holder of a corresponding authority token can request one. These keys are therefore secret and unique to a particular foundation on a given machine, but not unique to instances of that foundation. This key can be used for a variety of schemes that require a secret.

Public foundations

Often the contents of a foundation are not a secret. It may very well be known ahead of time what should be contained, and the role that the foundation serves is to give assurances

about the contents and that they cannot change. This could be implemented with a single entry point that will, on entry, derive a read-only capability to its initial segment and then immediately return. This self-exposing foundation will never release a capability with write privileges to its payload, and the nanokernel guarantees there is no other way to get to the foundation data. The entry token would be attestable as a foundation that performs this limited release, and is also callable to get a read-only copy of the data.

Because this pattern of a foundation exposing itself as read-only was found to be common, the nanokernel helps to facilitate it. When creating a foundation, it can be specified that the foundation is 'public'. This allows any holder of an entry token to the foundation to request from the nanokernel a read-only capability to the foundation's initial segment. Public foundations do not get the writeable capability on entry, if indeed they have an entry, which they often do not. This example illustrates how foundations differ from traditional enclaves. They are not necessarily a primitive for secrecy, which is achieved by combining with reservations, but are instead a mechanism for reasoning about how the contents of memory were constructed and what else might be able to access them.

Public foundations have a few uses. They are used in the dynamic verification of fast-leaf calls, as is discussed in section 5.1. Another use is secure deduplication.

Secure deduplication

In CheriOS's world of mutual distrust, each compartment is left to decide its own policy. 'Policy', in this regard, is just code. In order to not place trust in the OS, every compartment decides how it will achieve its domain transitions, exceptions, and security checks by way of containing code that implements the correct functionality. Although each compartment is allowed to decide its own policy, often the result is that every compartment will decide to do the same thing. Every compartment will use the, hopefully, secure set of handlers the toolchain providers design. Traditionally, a lot of this functionality could have been provided by the OS, or by shared libraries, but due to mutual distrust every compartment statically links in the same routines. This leads to bloated code and a degradation in performance.

In order to combat this penalty, CheriOS offers a secure deduplication mechanism built atop foundations. In order for a compartment to substitute its own pointer to memory with another, it needs assurances that the contents cannot be changed. This can be achieved by putting the data in a public foundation.

CheriOS contains a deduplication service that can be provided with either a SHA256 hash, or a copy of the data to be deduplicated. The deduplication service will return a public foundation entry token that the caller can verify is indeed a token with the correct SHA256 hash, and then use in place of their own data.

Now compartments are free to choose their policy, but can elect to just use the OS provided handlers, with assurances from the nanokernel that the contents of memory cannot be changed. If the OS promises to deliver a set of handlers, it is not even necessary for a compartment to include

the code it wishes to deduplicate, only the SHA256 hashes are needed. Currently, no programs in CheriOS use the lightweight mechanism of just storing a hash of common code, as there is a lack of compiler support. Instead, there exists a deduplication function offered by standard libraries that will, during initialisation, scan the entirety of a program's read-only segment, attempt to deduplicate the functions and read-only objects it contains, and then compact the segment in-place. Details of the implementation of this function can be found in appendix E. See section 5.4 for statistics on how much static data can be deduplicated with realistic programs.

Although CheriOS considers side channels to be the programmer's problem, such deduplication might alarm some readers. If a programmer were concerned about side-channels in a particular application, they can always disable deduplication, as it is purely an optimisation. The OS cannot force a program to deduplicate, the program must decide to do it. Furthermore, deduplication does not actually increase efficacy of side channels like cache timing attacks, which do not require sharing of addresses, only of cache lines (where multiple addresses will alias). Attacks that exploit shared micro-architectural state that use the entirety of the address, rather than just some of the bits, might be worsened by such deduplication.

Deterministic linking

System wide deduplication on an object granularity can remove a surprising amount of data. Strings, especially, tend to be duplicated between multiple programs. However, functions were the initial goal of deduplication, as CheriOS contains so many copies of the same code. After designing the deduplication pass it was noticed that results were not as good as promised. Programs that linked in the same static libraries were not successfully deduplicated. This was down to static relocations. Even if two programs link in the same function, even containing the exact same instructions in the same order, relocations will make the final binary different. The relocations that tend to differ between different linkings of the same library are indices into captables for global symbols. Programs only include the symbols they require, and have symbols of their own, and so order their captables differently even if they include a similar set of symbols. The order is, however, arbitrary, and so we might make a better choice in order to aid deterministic linking, which will in turn improve deduplication.

There is no deterministic placement strategy we could ever choose that would guarantee the same location for a symbol on every linking. This is easy to see by pigeon-hole principle. A static library may have many more symbols than the programs that link it in, and any subset could be used. It is therefore not possible for all of them to always have the same position. Instead, we adopt the following strategy: each library assigns each of its symbols a 'preferred' position. Libraries will give smaller preferred positions to symbols that are more commonly used, or based on some other likelihood-of-inclusion heuristic. During the link step, we then give each library a priority. Currently, this priority is based on order of dependencies between libraries, i.e. first priority is given to libc, then other libraries a program may link in, then the program itself. Priorities are assigned in this way as we are more likely to see success in deduplicating standard libraries, and so getting their symbols in preferred positions is more important.

We then assign symbols to a program's captable as follows: starting with the highest priority library, loop through each of the symbols that need inclusion. If the symbol can have its preferred position (i.e. it is not taken, and not outside the table), assign it to the table, otherwise leave it for now. Once as many symbols as possible are in their preferred position, we loop through the libraries again to place the remaining symbols in the gaps.

You might think of this as treating the captable as a hash table, inserting symbols at deterministic locations, and then filling in gaps where there are collisions to achieve maximal density. One could imagine more complex strategies that have analogues with traditional hash strategies. For example, probing more than one location (i.e. cuckoo hashing or linear probing), or increasing the size of the target table would both aid deduplication by ensuring more symbols get their preferred position. Neither of these were explored, opting instead for the simplest solution that kept density to a maximum.

This strategy of more deterministic symbol placement was implemented in LLD and is enabled by default when building programs for CheriOS.

Another strategy considered was of lifting offsets out of functions and promoting them to arguments, calling every function via a stub. This would result in the function proper always deduplicating, leaving only index lifting stubs in programs. Because of the complexity of implementation, and the assumed cost of extra register pressure, this option was not explored.

Call Type	Cycles	Instructions[5]	Cycle Overhead	Instruction Overhead	CPI
Standard	14.33 ± 0.01	4	1	1	3.58
Fast Leaf	15.5 ± 0.01	7	1.08	1.75	2.21
Nano Call	22.33 ± 0.01	7	1.56	1.75	3.19
Complete Trusting / Trusted	43.34 ± 0.01	25	3.02	6.25	1.73
Complete Trusting / Untrusted	87.71 ± 0.01	53	6.12	13.25	1.65
Trusting / Trusted	64.71 ± 0.01	35	4.52	8.75	1.85
Trusting / Untrusted	110.02 ± 0.01	63	7.68	15.75	1.75
Untrusting / Trusted	144.96 ± 0.65	92	10.12	23	1.58
Untrusting / Untrusted	190.87 ± 0.58	120	13.32	30	1.59

Table 5.1: *Round trip times for different function call types. Each sample averages 4000 calls.*

Transition Type	Cycles
Exception to ASM handler	215.00 ± 0.00
Exception to C handler	393.00 ± 0.00
Trusting Message Send	1060.67 ± 8.25
Untrusting Message Send	1277.43 ± 4.7

Table 5.2: *Round trip times for other transitions. Each sample averages 4000 transitions.*

5.4 Evaluation

Calling conventions

Micro-benchmarks

Here the cost of the domain crossings outlined in section 5.1 are measured in isolation from a larger application, using a bespoke micro-benchmark. Figure 5.1 shows the costs of the different styles of synchronous function calls. The costs are for both edges of a round trip to a void-typed function that does no work and returns. Every effort was made to keep other effects out of cycle costs. All interrupts were disabled for the test, and the paths were all warmed up before the benchmarks. Calls were made in batches of 4000 to reduce overhead from the calls to the timer functions made at the start and end. The loops themselves were manually unrolled enough so that loop control instructions were insignificant compared to the loop body. Significant additions, like instructions to pad delay slots in branches, *were* included in the instruction count as they would be executed on every call even after unrolling. It is also worth paying close attention to the CPI. The current CHERI-MIPS implementation was built for research purposes and should not be considered production grade. It suffers from many avoidable hazards, for example, always taking a several cycle penalty on a branch. As such, some of the cycle counts might seem a little inflated. For example, even a standard function call which is just one branch in, and one branch back, takes 14 cycles. A warmed up call on a modern processor should be able to correctly predict such behaviour without fail, resulting in CPIs closer to one. For this reason, the instruction counts have been provided for comparison[6].

Fast leaf calls (which offer the same security as Untrusting/Untrusted), and calls into the nanokernel, are particularly fast, being within a factor of two of a standard function call with respect to both cycle and instruction counts. These calls are expected to be made frequently. At least two nanokerel calls are needed for every `malloc` call, for example. The others all incur some costs due to crossing dynamic library boundaries as ABI related registers need saving and restoring regardless of trust. One might describe a traditional OS's syscall to be a 'Complete Trusting / Untrusted' call. CheriOS can achieve this in as little as 88 cycles (6 times that of

[5]Includes a nop in a delay slot on the call side that would normally be filled.

[6]After the measurement of these benchmarks a bug was found in the CHERI-MIPS FPGA pipeline that caused regular spurious stalls. These might have been behind the poor CPIs measured. Any comparisons between OSs will have used the same bugged pipeline.

a standard function call) with no hazardous exceptions. For only another hundred odd cycles (13 times the cost of a standard function call), full mutual distrust can be be achieved. It is worth noting where these extra instructions come from. The difference between a 'Trusting' and 'Untrusting' call is 57 instructions. A total of 34 of these are just for spilling and restoring registers, not for security checks. An architecture with support for spilling and restoring registers efficiently could make the two even closer in cost. Compared to a standard function call, the fully distrusting calls are still an order of magnitude more expensive. However, they are perhaps more comparable to the cost of IPC on other OSs, in the way they are able to facilitate communication with other software compartments. CheriOS can replace standard IPC with shared memory structures protected by the distrusting calling convention. For example, CheriOS's cost for distrusting calls is still several times lower than IPC costs on seL4, which cost on the order of a thousand cycles[7].[115][63] If a fast leaf call can be used, the difference is even more stark.

Table 5.2 shows the costs for two other types of transition: object activation message send and synchronous exception handling. Again, each sample was the sum of 4000 calls, and the time is for a complete round trip. In the case of message send this is the cost to enqueue a message[8], context switch to the target activation, read the message from the queue, call a target function that does nothing with the message, go back to sleep waiting on the queue, and context switch back to the source activation. As message send and message wait are both function calls into the microkernel, the costs for different trust modes are given. For untrusting both the source and target activation distrust the kernel. It should not surprise the reader that the extra cost for being untrusting is that of two times the difference between a 'Complete Trusting / Untrusted' and a 'Untrusting / Untrusted' call; one for the send one for the receive. A large component of the message send cost is that of two times the cost of a full context switch. CHERI-MIPS has ∼60 registers that need switching, resulting in ∼240 loads and stores for the switches alone. Like with spilling for distrust, an architecture with support for load and store multiple, register windows, or just fewer registers, might get the costs down further.

Two cycle counts for user exception delivery are given: one for handling in pure assembly and one for handling in C. In order to handle exceptions in C, an assembly handler changes to a new exception stack, pushes the registers the nanokernel did not, and then jumps to a C handler. The provided runtime handler allows for registering C handlers based on exception types. The handler for exceptions related to needing a new slinky stack safe segment are handled in this way.

Macro-benchmarks

Here, the impact of mutual distrust on a system wide workload is measured. The same HTTP workload as in chapter 4 is used again as it makes heavy use of system calls, the socket layer, and message sending, all of which result in domain crossings. Workloads that spend most

[7]The most comparable RISC-V board has a server-client round-trip of 1107 cycles, but exact cycle counts vary depending on architecture.

[8]CheriOS features a fast path for message send when there are no messages in the target queue and the target is already waiting on a message. It is likely that this is always hit in benchmarks, and so the cost of en-queuing would be reduced.

	Request Times / ms		
Mode	Mean	Median	Standard Deviation
Trust	3.181 ± 0.012	3.074	1.682
Distrust	3.427 ± 0.015	3.320	2.114

Overhead		
Mean	Median	Standard Deviation
7.74% ± 0.62%	8.00%	87.54%

Table 5.3: *Request times for 1KB file with and without mutual distrust.*

of the their time doing busy work in a single compartment are not interesting for the purpose of discussing the ramifications of enabling system wide mutual distrust. File and socket operations especially require coordination between compartments, involving the block-device, filesystem, network-stack and Ethernet-device compartments. A full trace of the system serving a file can be found in section 6.3.

We operate the system in two modes: trust mode and distrust mode. In trust mode, programs are loaded in a way that means the program loader could keep capabilities it should not. User programs always trust both system libraries and the microkernel. As it would not be sensible to suggest an OS trust its users, all OS components are still distrusting of all others. All memory is still allocated using reservations; this is a fundamental CheriOS property and is never disabled. However, the program loader could potentially have kept capabilities to the loaded program, and so could be exfiltrating capabilities that are the result of taking reservations.

In distrust mode we do two things. Firstly, programs are loaded in foundations[9] to ensure they have been correctly loaded and so that no other part of the system holds capabilities to their stacks or globals. Some parts of program loading are moved into the foundation to ensure they are done correctly. Secondly, all programs use the untrusting calling mode for all calls.

For both modes, temporally safe stacks are disabled in order to not conflate costs due to safety and distrust. See section 4.4 for how temporally safe stacks effect the benchmark.

Table 5.3 shows the time for the HTTP requests to complete in both cases, and the overhead for enabling mutual distrust. The overhead is, of course, very workload dependant. The same benchmark run with a 1MB file only suffers a 0.64% overhead as the same number of boundary crossing do much more total work. The 1K file size used in the benchmark was chosen to illustrate a worst case for the particular application.

Figure 5.4 gives a breakdown of how many calls of each type were made for a single HTTP request (averaged across 10 000 requests). The breakdowns are by application, not compartment.

[9]Some early loaded programs like the microkernel are not due to bootstrapping issues. However, all the userspace programs that do substantial work and handle any sensitive capabilities are.

Application	Call Type					
	Nano Calls	Fast Leaf	Trusting	Untrusting	Trusted	Untrusted
Microkernel	251 / 251	0 / 0	0 / 0	0 / 0	- / -	56 / 56
Block Cache	1 / 1	0 / 0	25 / 0	4 / 29	- / -	13 / 15
LWIP	8 / 8	19 / 19	39 / 0	3 / 42	- / -	22 / 25
FAT Filesystem	0 / 0	0 / 0	47 / 0	3 / 50	- / -	33 / 34
NGINX	20 / 20	0 / 0	42 / 0	4 / 46	- / -	37 / 37
Other	3 / 3	0 / 0	0 / 0	0 / 0	- / -	0 / 0
Total	283 / 283	19 / 19	153 / 0	14 / 167	6 / 0	161 / 167

Trust / Distrust

Table 5.4: *Call-type breakdown across applications for HTTP benchmark when running the system in both Trust and Distrust modes.*

For example, many applications use the socket compartment (which is always distrusting), and so even in Trust mode there are many untrusted calls the application makes. The main difference between the two system modes are in the number of trusting and untrusting calls. In Trust mode, 153 out of 167 calls are trusting, but all of them become distrusting in Distrust mode. Notably, very few trusted calls are made in either case. This is because most applications are only making calls from the main application into the microkernel and socket compartment, which never trust applications. The largest number of calls made by count are nanokernel calls, predominantly from the microkernel. Every time the microkernel needs to make a privileged action, it invokes a nanokernel capability. Recall from the microbenchmark the cost of nanokernel call is on the order of a standard function call.

Deduplication

Here, the evaluation of the automatic deduplication and compaction pass described in section 5.3 is presented. Before any program starts, the deduplication service is initialised to contain all of libc and the standard user libraries. Later, when programs are started, they attempt to find duplicates with the deduplication service, but do not create any new entries. This is not as aggressive as it could be (it will miss anything not present in libc), but will still highlight how much library code can be eliminated even with minimal effort.

[10]Skewed by a few images embedded in the program.

Program	Deduplications		Compaction	
	Functions	Objects	Total Size Size / KB	Reduction
Hello World	108 / 148	135 / 137	65.7	60.6%
Type manager	119 / 164	136 / 136	68	58.7%
Block Cache	128 / 179	142 / 167	77.4	52.5%
Block Driver	131 / 175	141 / 190	75.5	56.7%
FAT Filesystem	137 / 196	137 / 189	99.1	44%
NGINX	187 / 879	185 / 1239	441.4	11.6%
LWIP	167 / 313	142 / 281	663.4	7.5%[10]

Table 5.5: *Deduplication statistics for a selection of programs. Shows successful deduplication as a fraction of total potential targets, for both functions and other read-only objects. Also shows size of read-only segments and reduction due to compaction.*

Table 5.5 shows the results of deduplication on a selection of programs, including those we use for some of the benchmarks, and a few others that were of interest. The programs have been listed in increasing order of the size of their read-only segments. Read-only object deduplication manages to remove every object that comes from standard libraries. For small programs, this results in a significant reduction. The 'Type-Manager' contained no read-only objects of its own, and so every one was removed. Hello World only had 2 read-only objects of its own, which were the strings 'Hello World!' and 'Goodbye World!'. Function deduplication is not as perfect. Hello World almost certainly does not contain 40 of its own functions (in fact it only contains a unique main). The other 39 were functions are from libc, but due to remaining non-determinism in relocations they were different from that of the dummy program. As the program sizes increase, the ratio of successful deduplications go down (as would be expected as unique functions go up), but the absolute number of successes increases. This is due to two effects: firstly, more symbols are included by more complex programs and so there are more opportunities for deduplication. Secondly, as programs get larger, so do their captables, and so more symbols can have their 'preferred' position. NGINX, the largest program by count of symbols, manages to deduplicate a substantial number of functions. The standard libraries included had a total of 390 functions, and although it is unlikely all of them were included, this means NGINX deduplicates at least half the functions statically linked in.

The deduplication pass is fully functional in simulation, and programs operate error free. However, due to difficulties experienced with cache coherency on the CHERI-MIPS FPGA implementation, the performance benefits of deduplication could not be measured with micro-architectural fidelity. The pass was disabled for the purpose of all other benchmarks performed in this document.

5.5 Related work

CAP

The organisation of CheriOS shares many similarities with the Cambridge CAP OS.[134] They both utilise capability hardware, and share some of the same goals, namely to create a capability-based operating system with fault isolation due to increased partitioning, and with a non-hierarchical privilege structure (unlike that offered by privilege rings). However, CheriOS differs both due to the availability of the more flexible CHERI hardware, and in its attempt to provide mutual distrust between the OS and its programs. Although the CAP OS was internally partitioned, no attempt was made to separate its privilege from user processes as CheriOS does. The CAP OS could freely manipulate running programs.

Capabilities on the CAP were not as flexible as that of CHERI. Not only could they not be interspersed with data, but they were limited in what they could refer to. Capabilities could be direct references to virtual memory (like those of CHERI), but these could only be stored in a single central table, the *MRL* (Master Resource List). Otherwise, capabilities were indirect, placed in either a *PRL* (Process Resource List), or a capability segment. Capabilities in capability segments always referred to the current PRL, and entries in the PRL referred to either capability segments or the MRL, these cycles eventually always terminating in a MRL entry with a direct capability. Every CHERI capability is effectively a direct capability, albeit to a virtual address space.

Each process on CAP would have a single PRL. Within a process, a number of *protected procedures* would exist, identified with an *Entry* capability. Invoking an Entry capability would both change control flow, and modify some entries in the PRL, giving access to the capabilities to only to be used by the protected procedure. Different protected procedures with the same code were called *protected procedure instances*. The PRL can be seen as a dual to the DLS in CheriOS, where protected procedure instances are the equivalent of domains. However, there are some key differences. In CheriOS, the entire DLS is swapped on every domain crossing. Processes on CAP share most of the global state in the PRL. For example, both CheriOS and the CAP OS have an entry in the DLS or PRL for an entry point for exception delivery. This is switched on CheriOS, but not on CAP. This requires a trusted per-process handler on CAP if faults are to be delivered to the correct compartment of a program. Furthermore, to modify the PRL, a protected procedure that all other protected procedures trust must be invoked. The less centralised structure of CheriOS's DLSs is a result of CHERI's more flexible capabilities.

Another key difference is that the construction of a new PRL, as well as entry and exit from protected procedures, was handled on the CAP by microcode. On CheriOS, this can be done using the more primitive sealing and ccall instructions, and does not require any extra trusted logic. This means that programs on CheriOS can construct new DLSs, grant them capabilities, enter them, and internally modify them without a trusted third party. This is important to support mutual distrust.

CAP also differed in its attempt to solve illegal control flow. The CAP uses a trusted stack (managed by microcode) to ensure returns are made in order. CheriOS ensures correct control flow using just the calling convention described in section 5.1. Although CheriOS only enforces compartment-boundary CFI, it does not require that added complexity (and trust) of a trusted stack.

Although the CAP supported virtual memory, it did not use page tables. Indirection through the MRL allowed segments to be moved to and from backing storage. If a segment were not resident in memory, an *outform* capability could be placed in MRL, which would fault on use. Although outform capabilities contained information such as the disc sector that contained the paged-out segment, it was left to a trusted program to manage paging. Furthermore, the OS had complete discretion on the management of the MRL.

The ability to modify the MRL on the CAP grants roughly the same power as modifying page tables on CheriOS, that to effectively modify the contents of virtual memory without an explicit capability, by instead changing mappings. CheriOS uses the nanokernel to insert simply-enforceable checks on page table modification to offer guarantees of isolation to programs, even if pagetable management is malicious. There is no equivalent on CAP.

The dual communication methods of procedure calls (lightweight, low-level and strongly-coupled) and IPC message sends (heavyweight, an OS construct, de-coupled) are present on both CAP and CheriOS. Implementation differences exist, such as allowable message contents, queue structure, and direct availability to read the queue without calling into the OS, but these are all relatively minor. A larger difference is the CAP views message sending as a trusted mechanism. On CheriOS it is untrusted, and the facility to protect capabilities being sent via the OS (using sealing either directly or via foundation signing) is offered.

seL4

There are parallels between the capabilities used by seL4[116][72] and Barrelfish with those of CheriOS. On seL4, an *untyped* capability (representing all of physical memory) is given to the initial thread by the kernel. Untyped capabilities can be subdivided and eventually retyped, creating a new capability and disallowing derivation of new capabilities from the untyped capability.

Taking a CheriOS reservation is roughly equivalent to retyping an untyped capability in seL4. However, whereas on seL4 the resulting capability is still managed by the kernel (as there are no hardware capabilities), on CheriOS, the resulting memory capability is managed completely by hardware. Furthermore, sealing can be seen as a further retyping operation, from a memory capability to a custom user capability type. On seL4, such a retyping would again involve the kernel, on CheriOS this is still an architectural operation. Having most retyping performed by hardware significantly reduces the size of the software component of the TCB, and makes these operations cheaper.

The other major difference is that, on seL4, untyped capabilities refer to *physical* memory, and CheriOS reservations and memory capabilities refer to *virtual* memory. Although the seL4 microkernel does manage the MMU, it is guaranteed to manipulate it correctly, as the formal proofs of its operation include a model of the MMU. However, even though virtual mappings on seL4 are manipulated by userspace through the use of capabilities (just as they are in CheriOS), use of these capabilities does not enforce the same invariants that CheriOS does in order to provide reservations. This has some important ramifications. Both seL4 and CheriOS can use their respective mechanisms to have untrusted user processes provide memory to the kernel. However, unless a seL4 process trusts the software managing its virtual space, the use of a virtual address does not give the same guarantees offered by a CheriOS reservation. The vast majority of address space management on CheriOS is outside the TCB (as far as isolation is concerned), the only exception being the nanokernel checks that virtual pages do not alias in physical space. This allows a CheriOS process to ensure it is isolated with a much smaller TCB.

It is worth noting that seL4 untyped capabilities can be used to partition physical memory for use by different applications. The CheriOS nanokernel only offers capabilities to manage all of physical memory, or none. This is not a limitation however, a wrapper could easily be constructed to create individual capabilities to physical pages using sealed capabilities. As this wrapper can be placed outside the nanokernel, it should be, so as to maintain minimality. Currently, CheriOS uses a single memory manager that manages the entire virtual address space, and all of physical memory. Better separation into multiple managers, both for physical and virtual memory, is possible future work. Importantly, it is envisioned that no change to the security critical nanokernel should be required.

Unlike seL4, CheriOS does not keep explicit track of a capability derivation tree. The only exception to this is the 'parent' operation on reservations, to facilitate revocation. This is intentional. On CheriOS, new capabilities are derived on every single pointer manipulation. Storing a derivation tree would be prohibitive. The benefits to this are that capability management is further decentralised, and users can efficiently create and manage their own types. The downside is that revocation is more costly, requiring an expensive memory sweep as capabilities may have propagated anywhere in the system.

seL4 also uses messages passing for IPC, but unlike CheriOS and CAP, does not support a second faster mechanism to transition between security domains. Because seL4 does not use hardware capabilities, the kernel needs to translate them as they are passed in messages. Hardware capabilities allow capabilities to be passed directly in registers with no further translation, with the same cost as a function call. This allows for more efficient fine-grained protection, using protected procedures on CAP, and domains CheriOS.

seL4 represents a schedulable context with a TCB (thread control block). This is somewhat analogous to a CheriOS CPU context. However, on seL4, a TCB context can be externally modified with the appropriate capability, by reading or writing registers contained within. A CheriOS CPU context can only be scheduled. This is due to a requirement for mutual distrust, neither the constructor of the CPU context, nor its scheduler, are trusted. Because such external modification is disallowed, CheriOS delivers traps differently from seL4, by effectively carrying

out a domain transition within the CPU context, allowing the trap can be handled internally. seL4 delivers traps in the same way both it and CheriOS deliver asynchronous interrupts, via IPC messages.

KeyKOS and Eros

Many of the key ideas of KeyKOS[60][17] survive in Eros,[118] so although here we discuss the features of Eros, concepts such as keepers, space banks, domains (processes on Eros) and their constructors, were all present in KeyKOS as well. One fundamental difference between Eros and CheriOS is that Eros is persistent. In fact, the difference between main memory and backing storage on Eros is invisible to all but the kernel. This is very different to CheriOS, which must reconstruct all capabilities from a primordial capability each time it boots. One key reason for this is that CheriOS capabilities are implemented and enforced directly by the CPU, and therefore cannot exist in the same form when moved to secondary storage.

Eros describes a virtual address space as a tree of nodes. Each node holds a table of capabilities, either to more nodes, some other capability types, or to a page of either data or capabilities. To modify this structure, userspace must invoke a capability. This structure intentionally matches quite closely to a conventional hierarchical page table, and the Eros kernel handles translation between the two when a page fault occurs. Translation of a node tree into an underlying hardware format, and transparent paging, are all handled by the kernel. This is substantially more trusted logic than on CheriOS, which only guards modification of page tables, but leaves their management to userspace.

Although the Eros kernel is responsible for managing page tables, kernel resource allocation is done in userspace. Eros has userspace components known as *space banks*, which are responsible for resource allocation, deallocation, tracking, quota limitation, and certain security properties. Eros space banks enforce allocations are exclusive at the point of allocation (just as CheriOS reservations and seL4 untyped capability retypings are exclusive), however, they must be trusted to do so. Requiring correctness of some userspace components for isolation guarantees is more trust than is required on CheriOS.

Eros separates the system into processes (roughly analogous to KeyKOS domains). A process root contains a number of capabilities, including one to designate its address space, and one for storage of its register file when not running. An Eros process root also has a keeper, a capability that designates another process to be sent a message if the current process experiences a fault. Eros keepers in process roots are much like the exception entries in CheriOS DLSs, and they also share similarities in that they contain the root capability set available to given security domain. However, a CheriOS DLS is finer-grained than an Eros process. The exception entries result in a low-level CCall, rather a high-level message send. Furthermore, on CheriOS, DLSs are separate from CPU contexts (which an Eros process effectively contains) and object activations. A single CPU context will transition between multiple CheriOS domains as different functions are called. CheriOS domains are more simply transversed as they do not imply a change in address space,

and so do not require interaction with the MMU, as is the case when transitioning between two processes on Eros.

Like CheriOS and seL4, Eros can enforce, and verify, sandboxing of processes. At the point of construction, a constructor can verify a new process has limited access to resources by walking its capability graph, starting with the process root that the process will inherit upon creation. However, there is little facility for a process to verify it is isolated from its constructor (or other processes) unless it trusts the chain of constructors used to create it. In fact, an Eros process capability conveys the authority to examine its registers, and alter or change its address space. This is required, for example, so that a keeper can rescue a process from a fault. On CheriOS, CPU contexts and domains are all designed to be be mutually distrusting of their constructors. Mutual distrust is achieved using foundations, which rely on only the nanokernel for isolation, still allowing the effort of process construction to be handled in userspace.

Programming for CheriOS and other capability OSs Although CheriOS and the OSs discussed here all draw inspiration from capability-based models, the guarantees offered and requirements made on programmers to achieve them result in some differences. The extent to which CheriOS programmers interface with capabilities on CheriOS is intentionally somewhat flexible, so that existing applications can be run alongside those that are (partially) re-written to take advantage of CheriOS. Like the other capability systems, capabilities must be invoked for every system-level operation, although this can easily be hidden using wrappers in order to offer a more UNIX-like interface. However, by using capabilities more ubiquitously, further benefits can be offered to the CheriOS programmer.

Although not a requirement, toolchains on CheriOS use capabilities for each and every pointer. This has a visible impact on data structure layout, even when not interfacing with the OS at all. In most cases, programs designed to run on other UNIXs should simply be able to recompile for CheriOS, but it is possible for this level of intrusion by capabilities to break existing applications. The benefit of automatic use of capabilities by the CheriOS compiler is a guarantee of memory safety. CAP's capabilities could also be used to achieve hardware-enforced memory safety, although they cannot be interspersed with data as is done on CheriOS. On seL4 and Eros, there is no analogue of such a low-level use of capabilities.

Each of the capability systems use capabilities to isolate parts of the system from each other. Enforcement is organised by the OS. However, it still left to programmers to decide which compartments should have access to which capabilities. Security is only improved if programmers ensure that compartments lie on meaningful boundaries, and increased compartmentalisation has a performance cost, leading to a cost-benefit trade-off that the programmer must consider. On CheriOS, constructing, entering, and delegating capabilities to compartments is cheap due to architectural support. On seL4 and Eros, syscalls must be employed for any of these operations. The CAP procedure call is quite lightweight, but without a mechanism like CHERI's sealing, constructing compartments still requires the help of trusted code. On CheriOS, compartments can be constructed using a sealing operation, entry can be achieved with as little as a single

instruction, and capabilities can simply be passed in registers without any translation required. This offers a more practical opportunity for increased compartmentalisation with low overhead.

CheriOS also takes a further step of trying to offer mutual distrust between applications and the OS. Doing so requires a method of authenticating software compartments, to ensure no malicious action was taken by the OS during compartment construction, which is done on CheriOS using foundations. To truly take advantage of this, CheriOS programmers need to manually verify the signatures (or at least write tools to do so) of software compartments they interface with, before sharing any capabilities. These authentication steps are extra work that a CheriOS programmer must undergo if they want to remove the OS from their TCB.

Trusted execution environments

CHERI, combined with reservations and foundations, offers flexible trusted execution environments to programmers. Here we will look at and compare with a few others.

SGX The major security difference between technologies like Intel's SGX and a CheriOS foundation is the boundary of the TCB. SGX and CheriOS both trust the CPU and a small amount of software, but CheriOS also needs to trust its memory hierarchy. Attacks against DRAM that exploit physical faults are a heavily researched area, first reported by Kim et. al.,[71] these are generally referred to as rowhammer attacks. Rowhammer has been shown to be powerful enough to gain kernel access,[114] and can be leveraged by untrusted code, even from sandboxed Javascript.[56] This is a particular concern, not just for CheriOS which relies on the integrity of DRAM for foundations, but for CHERI machines in general where the effect of flipping a bit in a capability can have serious consequences. It is envisioned that any secure CHERI platform would have to be hardened against rowhammer. Using ECC and increased refresh rates, for example. Hardening against rowhammer is still an area of research, and attacks against even ECC protected DRAM are still being found.[2][27]

There are also other physical attacks possible when extending trust to the memory subsystem, including physically snooping on the bus, or cold boot attacks. CheriOS, and CHERI, are defeated by these attacks. However, these attacks are not considered a real practical concern in the CheriOS attacker model. If a user really faced an attacker that could gain extended, unrestricted, physical access to their machine they are better off investing in physically hardened devices such as TPMs to protect their secrets, rather than relying on technologies such as CHERI.

On the other hand, foundations offer many practical benefits over SGX. First, there is the overhead of operating the MEE (Memory Encryption Engine). SGX requires nonces to be stored with every encrypted block to protect against chosen plain-text attacks. These nonces have to be large enough to, practically, never run out. Furthermore, a merkle tree is required to cover all enclave memory for the purposes of integrity. For 128MB of enclave memory, 25% is required just to store the tree and nonces.[57] Not only does this increase bandwidth and memory consumption, but even worse, it can increase the number of round-trips to memory. In the worst case, a single access might require as many round trips as the merkle tree is deep, i.e. logarithmic in the size

of memory. In order to ameliorate this cost, Intel chips require a second cache for nodes in the tree; this in turn increases power and space costs on die. Reports of cycle costs of the MEE vary. Overheads for SPEC workloads run vary from 2% to 12%[57][11], and from 55% to 420%.[133] There are also costs associated with entering and exiting enclaves, so called 'ECALLs'. Work by Weisse et al.[133] on evaluating the cost of calling enclave functions reports between 8600 and 17000 cycles (compared to 150 cycles for regular syscall on the same system) to call an enclave function. These are much higher than the CheriOS micro-benchmark measurements in section 5.4. Even with all these high costs, SGX enclaves are inflexible. The total memory available is small and statically fixed at boot time, boundaries are coarse grained, and enclaves cannot be nested.

CheriOS foundations suffer none of these downsides. Aside from the metadata that stores the ID of a foundation (which is a constant number of bytes regardless of size), foundations have no memory overheads. Any memory can be foundation memory (as memory is only informally attached to foundations), and so memory need not be divided on boot. Enforcement comes for free on CHERI platforms, or rather costs no more than already using a CHERI CPU. Foundations can be arbitrarily nested. For example, if a program were loaded in a foundation, it could load a compartment in a further foundation that had access to the outer foundation by delegating an authority token. Foundations are more flexible in what data is a part of the foundation, and what permissions the outside world can have with respect to their memory. They are not an example of ambient authority, and so are not prone to confused-deputy attacks. Their authority can also be delegated, which creates a great deal of flexibility. Entry and exit is also practically free, requiring a minimum of a CCall and a load to get an authority token from a compartment that did not have access. These are far cheaper than an ECALL.

ARM TrustZone TrustZone is even more inflexible than SGX in providing a trusted execution environment to applications as it offers only only a dichotomous 'secure' and 'normal' world. There is nothing to protect the secure world from itself, and applications in the normal world are still at the mercy of the trusted OS, so TrustZone cannot provide the same level of mutual distrust as SGX and CheriOS foundations.

AMD SEV Like SGX, AMD's SEV has the benefit of moving the TCB to the chip boundary, whereas CheriOS relies on the integrity of memory. Like CheriOS's compartment model, SEV is scalable enough to isolate every VM, but is not granular enough to protect at smaller scales as CheriOS does. Most importantly, SEV does not provide any integrity guarantees, and so cannot provide robust compartmentalisation.

SEV-SNP, an upcoming extension to SEV, will have integrity guarantees. The white paper describes a similar MMU management strategy to CheriOS, ensuring a one-to-one mapping of pages that is not allowed to change. It should therefore offer similar security guarantees, but will still only function at a VM granularity as access is restricted based on address-space identifiers.

[11] Notably these are result produced by Intel themselves, and were run on quite a small subset of SPEC.

Sanctum Another hardware only solution, sanctum,[33] provides a strong, encryption-free isolation mechanism. Access to memory is controlled by the hardware page-table walker, which ensures entries to enclave memory cannot enter the TLB when not in enclave mode. This dispenses with the memory overheads of encryption, pushing out the TCB boundary to include the memory system. Sanctum has the same problems that SGX does with respect to inflexibility (pages either are or are not enclave memory, we are or are not operating in an enclave). Furthermore, sanctum requires completely shooting down the TLB when exiting an enclave, making frequent transitions impractical.

Donky Donky[112] takes a domain-identity approach to compartmentalisation. Every memory page is given a 10-bit key, and a domain is a set of authorised keys. Hardware enforces pages cannot be accessed without the appropriate key, and switches are achieved using traps and a software monitor. Capability systems are more flexible than domain-identities. Arbitrary bytes can be accessible in a given capability graph, rather than just 2^N flexibly-sized regions N-bit keys offer. Furthermore, capability systems are decentralised, and separate policy from enforcement. The Donky monitor centrally stores domain information, and must be able to handle any policy requirements of programs. The capability graphs in CheriOS allow individual domains to select which others may enter them, without any central facility keeping track.

Deduplication

Plentiful previous work exists on duplication at a file or VM level. Guest VMs often run similar OSs and libraries, so deduplication of common guest VM pages by the hypervisor can significantly reduce memory usage. This is commonly done by scanning guests for duplicates. Duplicate pages are eliminated using the MMU, marking the duplicate COW (Copy-On-Write). There are several implementations: KVM refers to it as KSM (Kernel Samepage Mapping)[11], while VMWare refers to it as TPS (Transparent Page Sharing).[130] CheriOS's deduplication technique is finer grained than KSM or TPS, as they must operate at a page granularity due to the MMU. Thus, CheriOS can find opportunities for deduplication that KSM and TPS cannot.

The security implications of deduplication among VMs are well studied. There are broadly two side channel to be exploited: the first a timing channel that comes from deduplicated pages being marked COW, the second an exacerbation of a standard cache-timing attack. If a running program is vulnerable to cache-timing attacks, then deduplication helps attackers in locating victim data structures; they load the same vulnerable program in their own VM. Using the COW channel, Suzaki et al. demonstrated the ability to detect what programs a victim was running,[126][127] and Xiao et al. established a covert channel[139] with tens of bits per second bandwidth. Irazoqui et al. demonstrated a cache-timing[64] attack on AES across VMs that exploited deduplication to locate lookup tables used by the AES implementation.

CheriOS does not use COW for deduplication (it only deduplicates read-only data and enforces this using capabilities), so does not have the first channel. CheriOS's deduplication does, however, reveal the address of any shared data, so does have implications with respect to cache-timing attacks. In CheriOS, data is deduplicated only if a compartment opts to do so, and

is done with knowledge of what data is being shared (the page-based techniques are blind to the contents). Compartments can therefore selectively choose to not share where appropriate. However, even without deduplication, it is a design principle of CheriOS that addresses should not be a secret. As data pointers are commonly shared between programs, it is too easy to infer a victim program's layout to keep it a secret. Cache timing attacks are out of scope for CheriOS, but a CheriOS-specific solution would need to not rely on secrecy of layout. If such a solution were in place, deduplication would have no further security implications.

Deduplicating programs on a program-symbol granularity has been explored less, as doing so securely is more challenging. Work by Will Dietz et al.[43] does so statically. In this work, multiple applications are linked into a single binary (deduplication is performed in this step), and then the binary is invoked with a switching argument to execute the correct program. Assuming the OS can be trusted, this is secure as only read-only pages are shared. It also avoids the problems faced on CheriOS to do with deterministic linking, as there is only one program being produced (without a GOT at all).

Work by Christian Collberg et al.[28] deduplicates dynamically, in a similar way to CheriOS. It does so at the granularity of pages, however. It makes the claim that offsets in GOTs are the same between different links. With true static linking we want to remove functions, merge GOTS, and potentially perform link time optimisations such as inlining.

CheriOS's deduplication is completely dynamic. A separately compiled program can be loaded and still deduplicated. The granularity is at a byte level, and so even if only a single function is included from a library, deduplication works. It also deduplicates not only the backing memory, but the virtual space as well, offering benefits to TLB pressure the other implementations do not. CheriOS also offers some solutions to making linking more deterministic. All of this is also offered with distrust of the OS, which the other implementations cannot provide.

5.6 Conclusion

This chapter has shown how CheriOS uses CHERI capabilities to enforce parts of the ABI that would normally have to be trusted. This model allows compartments to distrust not only each other, but most importantly, parts of the OS they would historically be forced to trust.

CheriOS uses a security hypervisor, the nanokernel, to provide additional enforcement and capability types. Among these are reservations, which compartments use to acquire memory with integrity and confidentiality, and foundations, which provide authenticity. Foundation primitives can be used in a similar fashion to cryptographic keys, allowing system interfaces to be treated as untrusted communication channels.

Chapter 6

Secure capability-based single-address-space IPC

IPC (Inter-Process Communication) allows processes to share the data they manipulate. CheriOS drivers and system services (such as filesystems) are just processes. Because of this, CheriOS does not distinguish between system-level IO primitives (files and network sockets) and IPC, the same mechanism is used for both.

CheriOS attempts to provide IPC with mutual distrust. IPC, by its nature, involves the movements of large amounts of data between multiple parties that have different access requirements. For example, reading a file necessarily involves the filesystem even though the filesystem neither owns nor requires the file contents. CheriOS processes require the capacity to enforce access control for data they send via IPC, and to authenticate the IPC endpoints receiving that data.

Traditional mechanisms (such as UNIX domain sockets) facilitate IPC by copying data between address spaces. Also, when underlying communication channels cannot be trusted, encryption is typically used to secure data. Excessive use of copying and encryption to facilitate communication between distrusting software compartments would create a tension between finer-grained compartmentalisation and imposing minimal overhead. Traditional monolithic OSs already make special effort to avoid overhead due to moving data, using interfaces like sendfile on Unix to avoid unnecessary copying where it can be avoided. Any replacement mechanism would need to be just as performant to see any adoption.

In this chapter I describe the CheriOS socket layer (section 6.1), a custom API for IPC, giving its design, implementation, and security features. CheriOS sockets are an end-to-end IPC mechanism that allow multiple mutually-distrusting compartments to organise the IPC control plane, communicating with each other and drivers, while offering authenticated access control for accesses to the actual data. Data structures normally confined to just the OS or individual processes are instead made to span across compartments and applications. CheriOS manages a single (although potentially disjoint) IPC graph, where end-points can either be arbitrary software, or drivers which perform real IO. Manipulation of this shared data structure is secured

using CHERI and the nanokernel, not the OS, so plaintext data can be sent without a loss of security. This allows compartments to share data buffers (or even logic) without copying, without unnecessary context switches, without having to share data with the OS, and with attestable control of where data is being routed (even if the compartments that organise the control plane are malicious).

Often, end-points for IPC are device drivers, which need to perform IO (Input/Output). The socket API is designed to fit naturally with device drivers, allowing secure lightweight sharing of buffers between processes and drivers for scatter/gather, further reducing the need for copying. Section 6.2 gives a brief description of how device drivers are written for CheriOS, and also discusses the challenges faced with offering secure DMA (Direct Memory Access). I evaluate this approach with a full-stack case study, using NGINX (a commercial web-server) as an application workload, demonstrating how, by taking advantage of hardware capabilities and a single-address-space system, efficient-yet-secure IO can still be achieved. As a point of comparison, I compare it to running NGINX on a more conventional system, CheriBSD (a port of FreeBSD with modifications to run on CHERI). Finally, section 6.4 offers a comparison of CheriOS's IPC mechanism to that of existing hypervisors and single-address-space systems, which show similarities.

6.1 Socket layer

Conventional systems separate processes and kernels into their own address spaces and enforce this separation using MMUs. Within an address space, data can be efficiently shared between different logical components by passing pointers to complex, page-spanning, fragmented data structures. Integration can be achieved by passing function pointers, used as 'callbacks'. Multiple address space prohibit the kind of object-granularity sharing and protection across process boundaries we find common within a process. Pointers have different interpretations in different spaces, and so portions of the virtual address space must be aligned ahead of time, or else some translation is required. Interfaces that involve callbacks do not function as code-pointers to unshared regions cannot be passed, or else a terribly insecure shared page of code must be used. Pages only allow coarse-grained sharing. Processes must organise for shared data structures to not share pages with unshared ones. Sharing a page that was initially unshared also requires system calls to modify page table entries. Because pointers to data structures are not shareable, the result is often to serialise data into a flat format. This is inconvenient, as it involves programmer effort, and has an implied performance cost. The result is that programmers try to avoid crossing the process boundary where possible, whereas it might otherwise be preferable to separate logic, both for a clean interface, and for security.

On CheriOS, these problems vanish. Pointers have the same interpretation in all processes, object lifetimes are manageable across processes (see section 4.3), fine-grained access control to sub-page objects is possible, and access can be granted as easily as passing a pointer (without OS involvement). This allows exposing data structures as they are within an application or the OS, rather than serialising and copying.

We propose that modifying shared objects is a good interface, and to illustrate the flexibility possible we showcase and go into the technical detail for one such structure, the CheriOS socket. Like Berkeley sockets, the CheriOS socket is designed to accommodate stream-oriented data transfer. Unlike Berkeley sockets, CheriOS sockets are asynchronous in nature and are designed for efficient communication between multiple compartments using scatter/gather descriptors. They are designed to require fewer context switches, and to enforce fine-grained access control that excludes the microkernel and other untrusted OS components from accessing data streams.

Initial exchange of shared-memory structures for the socket layer is still done using conventional microkernel message-send (which can be protected using sealing or other CheriOS primitives if desired; see section 5.3).

Shared memory request queues

A key design goal for CheriOS sockets is to avoid copying data. Unlike a traditional pipe or socket, the CheriOS socket primitive is not defined in terms of a reader and a writer. This is because reader/writer semantics do not capture the notion of buffer ownership sufficiently. Instead of framing IPC in terms of the flow of data, CheriOS does so in terms of the flow of capabilities. Each socket has a *requester* (section 6.1) and a *fulfiller* (section 6.1), with a different interface for each side. Capabilities always flow from the requester to the fulfiller, and represent a temporary transfer of ownership from the requester to the fulfiller. There are two types of socket:

Push In a push socket the writer is the requester. The requester sends read-only capabilities that point to the data to be sent. The fulfiller then reads from these buffers. The direction of flow of capabilities is the same as the direction of flow of the data.

Pull In a pull socket the writer is the fulfiller. The requester sends write-only capabilities to empty buffers. The fulfiller writes data into these buffers. The direction of flow of capabilities is the opposite to that of the data.

Push and Pull semantics allow us to map more naturally to different IO types, and often allows less copying by being more flexible in where buffers must come from. For example, TCP sockets on CheriOS are push in both directions, as data is sent at the behest of the writer, but arrives even if an application makes no request for it. Berkeley sockets, and their read/write interface, would be described as push-write and pull-read, as in both cases the buffer is provided by the application making the request. File sockets on CheriOS are push-write and pull-read.

Figure 6.1 shows an example IPC graph in CheriOS when running a number of applications that require both file and network access. As can be seen, for any particular application, data has to go through several mutually distrusting compartments. Most compartments are not sources or sinks for data, instead they simply redirect data to another compartment. System abstractions like filesystems and network stacks are entirely like this. Applications such as NGINX perform both data routing and are also a data source.

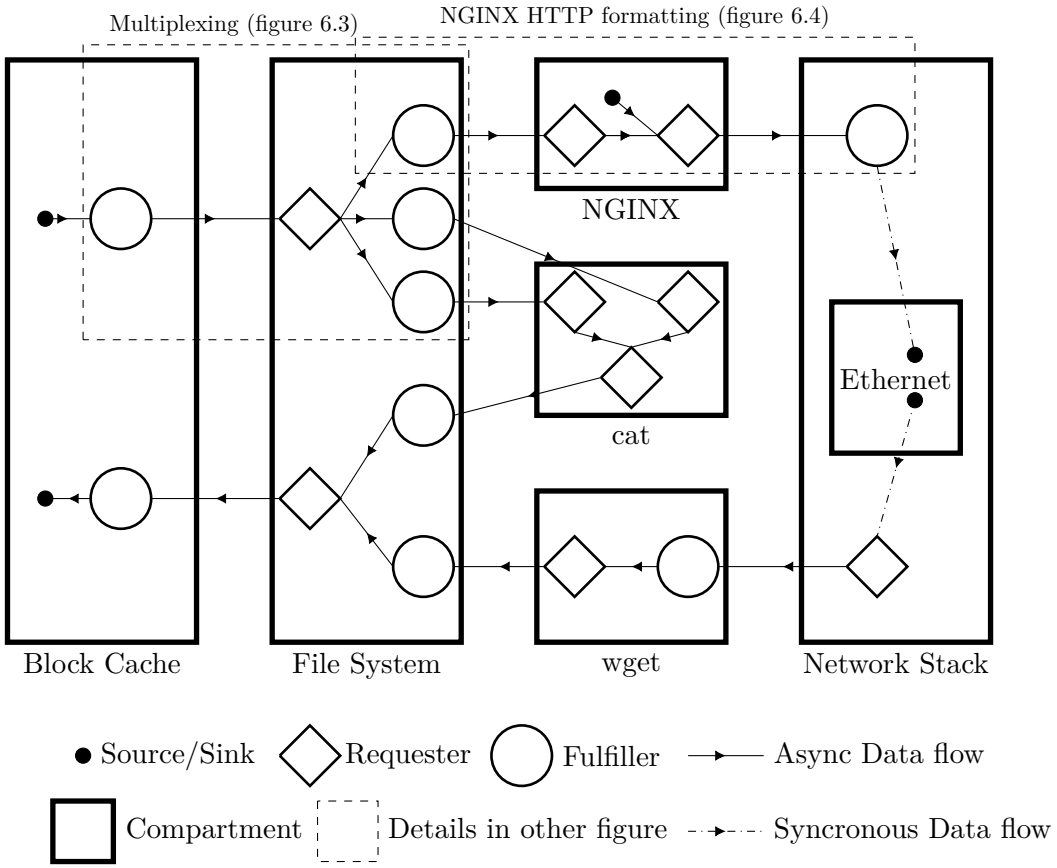


Figure 6.1: *An IPC graph in CheriOS. Adjacent / nested boxes represent inter/intra-process compartment boundaries, respectively. The dashed boxes show sections of the diagram expanded in other figures.*

Field:	Length	ADRB Inc	Type	Payload
Bytes:	8	4	4	16

Figure 6.2: *Socket request layout*

CheriOS sockets are designed so that compartments can set up IPC graphs such as the one in the diagram. Once set up, only end-points need scheduling. End-points are able to walk the graph and access buffers directly, without having to switch contexts or copy data, but are constrained by CHERI to do so properly. For example, the network can walk the IPC graph to access data buffers from NGINX and the block cache directly. The network stack's use of the graph is mediated by the socket library compartment, and so is unable to walk parts of graph not intended for it. Furthermore, compartments can restrict which source/sink a socket is eventually connected to, even if the system inserts several layers of abstraction. For example, NGINX requests that the Block Cache be the source for its data, and the Ethernet driver be the sink, even though it is directly connected to neither. The following sections outline the interfaces for requesters and fulfillers, and shows how sockets are connected together to create IPC graphs.

Requesters

A request queue is a ring-buffer-based single-writer single-reader queue of fixed size requests to transfer an amount of data. Like most queues of this ilk, they contain a header that includes the two indexes into the ring buffer: the requester's head index, and a fulfiller's tail index. Requests themselves are placed into a single queue, at the requester's head index, and are dealt with in a FIFO fashion starting at the fulfiller's tail index. The requests in this queue are much smaller than the OS's IPC messages, only 32 bytes compared to 128 bytes. Furthermore, rather than being up to the interpretation of the receiver, each field has a pre-defined meaning, see figure 6.2. Each request has a capability sized payload, a type, a 64-bit length, and an 'ADRB increment', which will be detailed later. The types of request important for understanding CheriOS sockets are:

Immediate An *immediate* request indicates the payload is the data to be transferred, and can be used for very small transfers up to 16 bytes. A classical ring buffer as they are used for data sharing could be considered to only have immediate requests.

Indirect *indirect* requests, the most common request type, provide a capability to a buffer to be read or written. Scatter gather lists, as commonly found in other interfaces, are captured by indirect requests.

Proxy A *proxy* request allows chaining together sockets without copying either requests or the data itself.

Join A *join* request is much like a proxy request, but connects to a different end of a socket. Both proxies and joins are intended to capture the forwarding of data from one socket to the other (for example, to implement sendfile), and are described in better detail in section 6.1.

Out of Band *Out of Band requests* are handled specially by fulfillers and are to allow for requests such as flushing or seeking.

Each requester will have a paired fulfiller, and together they form a socket. All request queues are managed by the socket library, which is its own compartment. In order to construct a new queue, applications must provide a reservation to the socket library, which will return an architecturally sealed object. This ensures that only the socket library has direct access to the memory used for sockets. All operations, such as making a request, must therefore be made by calling into the socket compartment. Applications are unable to modify sockets directly; this allows the socket library to ensure that sockets are managed properly. Almost every program will link dynamically with the socket library, and although they are free to trust it, the socket library is always distrusting of other compartments. Sockets created by the socket library can be shared by an application with other applications, and these objects will be usable by those applications in conjunction with their own link with the socket library. This is possible due to different instances of the socket library sharing a sealing capability, and the use of a SAS.

Data transfer on CheriOS sockets is always asynchronous. Indirect requests give temporary ownership of the payload buffer to the fulfiller. Until the fulfiller fulfils the request, the requester needs to not use the buffer. Failure to do so may result in writing undesired data, reading junk data, or even having the socket library close the socket. Asynchronous transfers can take large and unbounded lengths of time. For example, if no copying is specified, a CheriOS TCP socket will only mark a request as fulfilled when the data has been ACK-ed by the machine on the other end of the network. This allows for a reduction in double-buffering beyond what conventional systems offer, but puts a lot of pressure on requesters to keep data constant. We discuss in section 6.1 how the socket layer helps if data needs to be copied. On the other hand, exposing this allows CheriOS to be truly zero copy when desired. Transfer to and from devices can be done directly to and from application buffers.

Although CheriOS sockets are asynchronous, requests are made and fulfilled with synchronous function calls into the socket compartment. It may be desired to wait for outstanding requests to finish, or wait for requests to be made. Blocking and non-blocking operations on CheriOS sockets refer to whether an application should block waiting on the request queue, not on the final transfer of data. Blocking on an event is achieved in CheriOS by granting special notification capabilities. A *notification capability* is a sort of object activation capability that grants the right to the bearer to notify the target activation. The microkernel offers a system call which will wake up an activation specified by a notify capability, only if the activation is sleeping on a notification event, leaving more complicated mechanisms to be implemented by standard libraries. This is the only way in which the microkernel is involved with sockets, and the mechanism is not specific to sockets. Activations can sleep waiting for any set of a timeout, IPC message arrival, or notification. When a party wants to wait on an event for a request queue, it writes its notification capability to the header of the request queue in a special slot. Like any other capability, these tokens can be delegated, and so different wake conditions can be built atop sockets by transferring the notification capability.

Fulfillers

Fulfillers do not get direct access to either the request queue or requests as individual events. Instead, there is single 'fulfil' function that can be called on a socket. When a program calls fulfil on a socket, it specifies some flags, the number of bytes it is willing to fulfil, an argument to be passed through to the callback, and up to three different callback functions. The socket library handles the complexity of walking of the request queue, visiting individual buffers with the callback functions as appropriate. None of the request types are visible to the fulfiller, only a flat series of memory buffers.

There are currently 3 callback functions, although not all of them have to be provided. The main callback function is used to handle the normal data stream. It will be called with a capability to a buffer, a length the application should try fulfil for that buffer, the offset into the data stream this buffer represents, as well as the additional argument that was passed through. The callback function the application writes should return how many bytes it managed to fulfil, or an error code (which has an implicit 0 bytes handled). This allows fulfillers to push back. If for any reason they cannot fulfil some bytes, fulfillers can return how many bytes they did

manage to fulfil and the socket library will handle the rest. Requests can be partially fulfilled, either because fulfil was called with a length shorter than a request, or because of pushback from the callback function. The socket layer handles tracking of partially fulfilled requests internally. Fulfillers need not be concerned with how many requests are being handled; they only deal with scattering or gathering to a series of buffers. For example, if an application calls fulfil with a length of 100, they may get callbacks with pairs of (offset, length) of (0, 21), (21, 39), and (60, 40). If the application returned 21, 39, and then 30, then 10 bytes would be pushed back.

The second callback function works in the same way as the first, but is for out-of-band requests. Both the type and payload are provided to the callback directly, and applications can handle them however they wish, once again returning a length of how many bytes were fulfilled. This is 0 for most OOB requests, but OOB requests may have a length if they can be partially completed.

The final callback is an optimisation, and is only used in conjunction with join requests. If a fulfiller is copying data from source buffers to fulfil requests, and would be willing to provide the buffer directly instead of copying, the third callback allows the fulfiller to return a buffer rather than the data bytes themselves. This is described in more detail in section 6.1.

The benefit of this callback-based IPC is that copying is reduced. Rather than reading data into a local buffer and then performing some function, fulfillers can perform this function directly on the fragmented source data. For example, when reading a TCP socket, a programmer will be given a series of buffers that are actually a subset of the buffers into which packets arrived. This kind of scheme is only practical because of a combination of a SAS and fine grained access control. Furthermore, as buffers to both the stack and heap are wantonly shared between programs for IPC, it is important to have temporal safety to avoid unintended sharing. Only because CheriOS provides all these things is the IPC mechanism tractable. The socket library handles the potential shared-memory race-conditions for en-queuing and de-queuing requests. However, even if they should not, it is possible for readers and writers to make concurrent modification to buffers. As what data being sent is always at the writer's discretion, it is of no security concern that they can change it at the last moment. But, for this reason, readers should not assume that the data in buffers is stable. How the socket layer facilitates copying, if desired, is discussed in section 6.1.

The flags provided to the fulfil function change how fulfilment operates, and can be used to achieve a number of paradigms. The most important flags that can be specified are:

- F_DONT_WAIT
- F_CANCEL_NON_OOB
- F_SKIP_OOB
- F_SET_MARK
- F_START_FROM_LAST_MARK
- F_PROGRESS

- F_SKIP_ALL_UNTIL_MARK

The F_DONT_WAIT flag is used to select either blocking (implicitly using the microkernel notification mechanism) or non-blocking operation. F_CANCEL_NON_OOB and F_SKIP_OOB are used to stop if an OOB request is found, or just skip it respectively. These are useful to peek ahead at either just the data stream, or just the OOB stream. Other requests can be skipped by just leaving the main callback function NULL. The other flags are used for better supporting asynchronous operation, to which we now turn our attention.

Consider the case of a block-device driver fulfilling requests to write blocks of data. The driver needs to peek as much as it can into the request queue, specifying a callback function that writes an entry into a device's descriptor ring for each buffer. However, the driver does not want to mark any of the requests as fulfilled until the device has signalled IO has finished, lest the application change their contents. In the meantime, the driver may receive further requests that could be translated, and does not want to block operation behind previously half-handled requests.

In order to support this, fulfillers have a *checkpoint* mark into the request queue, as well as an official position in the stream. This checkpoint is notionally a mark in the data stream where the fulfiller last peeked to. If F_SET_MARK is set, then the socket library will set the checkpoint mark to wherever the IO gets to during the operation. This is possible to track as applications report exactly how many bytes they consume / produce in their callback functions. F_START_FROM_LAST_MARK will start fulfilling not from the tail, but the mark that is set by F_SET_MARK on the previous call when it was specified. F_PROGRESS specifies whether or not requests should be marked as fulfilled, not specifying it results in a peek into the queue.

To go back to the example of the block driver: when new requests come from the application, the driver specifies F_DONT_WAIT | F_START_FROM_LAST_MARK | F_SET_MARK, with an infinite number of bytes specified as the length to fulfil. This will result in resuming from the last buffer the driver transcribed to a descriptor, reading as many bytes as possible, but not marking any request as fulfilled. It will also set the mark for the next time we call fulfil in this way. When the device notifies the driver that X bytes have successfully been transferred, the driver calls fulfil with F_PROGRESS | F_SKIP_ALL_UNTIL_MARK and a length of X. This will cause the socket library to set the appropriate amount of requests as fulfilled without re-processing them, this time starting from the start of the stream. The mark will be left undisturbed.

F_PROGRESS cannot be used in conjunction with F_START_FROM_LAST_MARK. Requests must be fulfilled in the order they are requested. If a device completes out of order, then the driver must handle re-ordering, or use more than one socket. Checkpoint marks also cannot be used to go back in the stream: if the mark falls behind the start of the stream, it is updated to point to it.

Auxiliary data ring buffers

The asynchronous nature of CheriOS sockets can be difficult to deal with. A requester may want their buffers back immediately, or for security reasons not want to release a capability they own. A requester could always wait for all requests to be complete in order to be allowed to use their buffers again, but in some cases the requester may be unwilling to for performance reasons, if round trip delay is very high. Instead, the requester needs to make use of copies of the data, and insert an indirect request to an auxiliary buffer which can be loaned out instead of the source buffer. Rather than have applications implement and manage this themselves, the socket layer makes this a part of the interface. Every request queue has an optionally attached auxiliary data ring buffer (ADRB). The header for the request queue contains the capability to increment the tail pointer of the ADRB. The ADRB is another single-writer single-reader queue, but full of data bytes rather than requests. If a requester, either push or pull, wants to not use capabilities to private data structures they can instead allocate space in the ADRB.

Space is deallocated (and made available for re-use) automatically by the socket library when a request in the main queue that has an ADRB increment is fulfilled. For example, a push writer can allocate space in the ADRB for the data they want to write, copy it in, and then create an indirect request to the ADRB with an ADRB increment that will deallocate the copied data when the request has finished. The ADRB increment of a request need not be equal to the length of the request. The length of the request is how many bytes should be read from the ADRB, the ADRB increment is how many bytes should be freed. This is useful, for example, to align data in the ADRB to facilitate faster copying. Space can be allocated in the ADRB that includes padding, and an ADRB offset specified in a request that is the length plus the padding. The payload of the request would point to past the padding, and have a length of only the length of the data. Pull sockets can also make use of ADRBs; they just copy out of the ADRB after the request has finished, rather than copying in before making the request.

Applications rarely handle ADRBs manually. Instead, they will make a pseudo 'adrb-indirect' request. The socket library will handle the copy from a provided buffer to the ADRB, and then create an indirect request to the ADRB rather than the input buffer. However, the socket library also directly exposes the ADRB to the requester for manual use. For example, CheriOS's fprintf function formats data directly into the ADRB of a socket, and when it gets too full, or a newline is reached, it will create a single indirect request.

ADRBs are created automatically for the UNIX interface wrappers for files and sockets. In order to not expose the asynchronous nature of CheriOS sockets, the UNIX wrappers will either block on every request until completion, or will always use the ADRB to make supplied buffers available for immediate reuse.

Objects put in ADRBs, unlike nearly every other object in CheriOS, are not temporally safe. Although the buffer underlying the ADRB is temporally safe, different objects will be placed in the ring buffer over time, and capabilities to it are released outside the socket library to fulfillers. This loss of safety is due to layering, an insecure allocator has been built on top of a secure one (something which is still possible to do in CheriOS), in order to improve performance.

This trade off is considered acceptable, however, as sockets are one to one. In the case there is only one end-point that uses the buffer, it is safe to reuse it for multiple objects. If more complex proxying is taking place between multiple different parties with different trust relationships, then applications have to take care to attach a new ADRB every time a socket is re-routed. In practice, this tends not happen. ADRBs are relatively rare, and in all the cases they are currently in use in CheriOS; there is only one endpoint in mind, and so it does not matter that capabilities to the ADRB can be stashed by the fulfiller.

More generally, when considering writing a custom allocator, CheriOS programmers have to balance the possibility they are introducing safety issues at a higher level, with an estimated performance benefit. In most use cases it should be true that malloc (which is safe) is sufficient for use. System-wide IPC is one case where it is sensible to make the trade off in favour of performance.

Proxying

All request types discussed so far have been concerned with transferring data between two parties. It is the goal of CheriOS to compartmentalise as much as is possible, but as we add more compartments we also risk adding more copying to transfer data between them.

Imagine the set up of NGINX running on CheriOS, serving files from disk over a TCP session. With the current level of compartmentalisation there are compartments for the block driver, the filesystem, the web server, the network stack (potentially with a compartment for TLS), and the network device driver. For each of these hops, we create a CheriOS socket. In order to link these sockets together, there a few options. We could copy the data itself, which would be slow, or we could copy requests forward. Either way, this would result in a ripple of context switches as we move along the chain. This seems unnecessary, as for the most part we knew ahead of time compartments would just be forwarding data, potentially interleaving some of their own or skipping some. A better solution would be to set an IPC DAG ahead of time, with only the source and sink having to be scheduled after set-up.

This problem is also faced by traditional OSs, but to a lesser degree. Special system calls are offered (sendfile on UNIX, TransmitFile on Windows) to speed up IO transfers that go from the OS, to the application, and back to the OS without scheduling the application. Sendfile on FreeBSD, for example, can queue up an IO operation that will perform a gather, then a copy from a file, then another gather, only when sending from a file to a stream. As it creates more compartments, CheriOS tries to solve this problem more generally, setting up long IPC chains where only the endpoints need be scheduled. CheriOS is able to queue up an arbitrary number of forwarding operations, interleaved with any others, between anything that uses the socket interface even if they have not had special implementation to support it. Furthermore, once set up, only endpoints need to be scheduled.

To facilitate this, there are two request types: proxy and join. A *proxy request* is a request to go to some other request queue and fulfil requests from there instead. A join request is a request to go to another request queue and place requests there. Think of CheriOS sockets as cables with

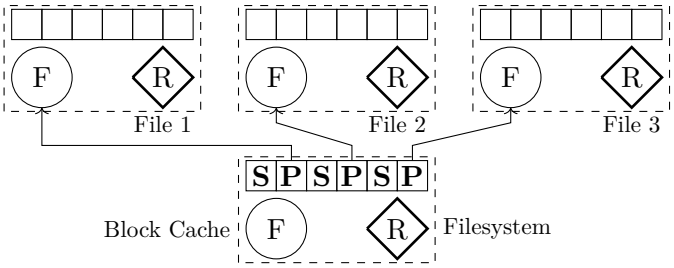


Figure 6.3: *Using proxy to multiplex disk access for multiple files. 'S' is seek. 'P' is proxy.*

a male (fulfiller) and female (requester) end. Proxies connect male to female, and joins connect female to female by placing a shim (a join) in one and connecting it to the other. Proxies and joins both describe forwarding operations, but proxies target the fulfil side of a socket, and joins target the request side. Proxies are used to connect one requester and one fulfiller, and so connect sockets of the same type, Push to Push or Pull to Pull. A proxy request may lead to another queue with another proxy request, and so the chain can grow arbitrarily long. Join requests, on the other hand, join a requester to a requester, and so can join Pull and Push together. A join request is always put in the pull requester, specifying the push requester as a target. There is no way of connecting two fulfillers without a copy as both endpoints will have provided their own buffers, one as a source and one as sink. If a process wants to forward between two fulfillers, they have to do so with their own CPU cycles, although the socket library contains a function to help facilitate this.

In figure 6.3, we show how the filesystem multiplexes access for multiple files onto a single socket it shares with the block cache, using proxies. Each open file is a socket, where the application is a requester and the filesystem a fulfiller. The filesystem is the requester for a socket it shares with the block cache, which is in turn the fulfiller. When an application makes a request on one of the file sockets, the filesystem first works out which blocks the request would correspond to. It then interleaves out of band seek requests (S) (to direct the block cache to the correct sector) with proxy requests (P) (that point to the file that made the request). At this point, no file data has been transferred, but as the translation from file to sectors has been completed, the filesystem can sleep. The block cache can fulfil file requests directly with no knowledge of how the filesystem works. The filesystem can multiplex file IO operations onto the single socket using proxy requests that point to different sockets. The block cache only sees a single stream of seeks and buffers. Multiplexing like this can get in the way of out-of-order completion as socket streams are completed in order. It is always possible to use multiple CheriOS sockets to allow more asynchronous operation. All CheriOS drivers, as well as the block cache, are able to handle multiple sockets if this is desired. For example, if a device supported NCQ (Native Command Queuing),[38] we would create a CheriOS socket for each tag in use.

A proxy request requires specifying a fulfiller, and a join request another requester. Until the requests are fulfilled, proxies and joins temporarily change who is the single reader and/or writer of a socket, and the socket library enforces this. If an application tries to use a socket they have proxied to, they will either block, or, if F_DONT_WAIT has been specified, they will get an error code specifying the socket is still being proxied. Internally, the socket library fulfils proxy

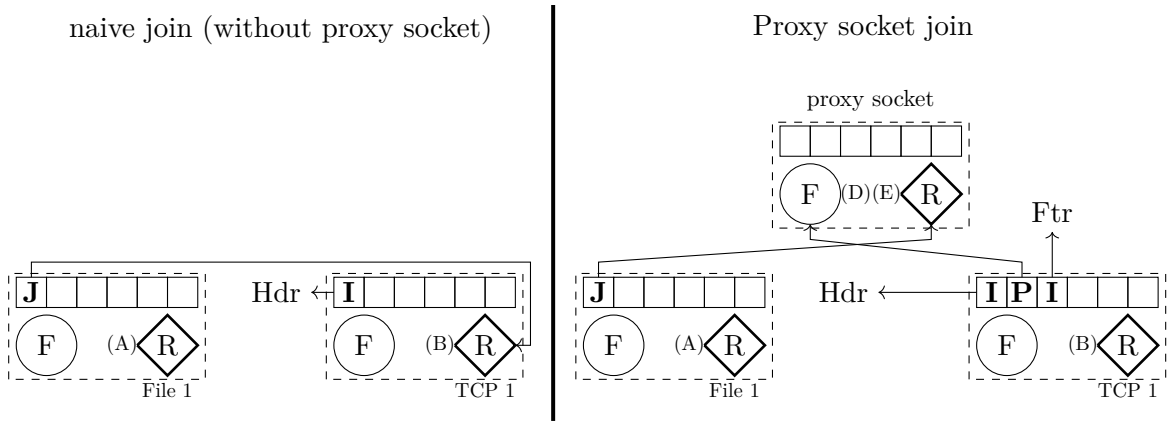


Figure 6.4: *NGINX using proxy and join to format a http packet. Naive join would block (B) from making further requests until join is complete. Non-blocking join uses a proxy socket. 'I' is indirect. 'J' is join.*

requests by recursively calling fulfil on the queue they proxy to, and so fulfillers will be unaware that proxies are even occurring. They still see a single data stream coming into their callback function.

To fulfil a join is more complex as they result in creating requests, which fulfillers do not do. The default way this is done is that the socket library allocates space in an ADRB to create a buffer, calls the normal fulfiller's callback to fill in the buffer, and then creates an indirect request to the ADRB in the queue that is the target of the join. If joining is handled this way, the fulfiller is once again unaware of joining taking place, and only needs to handle scattering/gathering. This might result in an unnecessary copy, but requires no change in interface. However, in the case that the fulfiller already has a buffer for the data, the third callback can be used. This callback is allowed to provide its own buffer, but still gets to specify how much data it contains. Fulfillers need not provide this function; for example, if they are generating the data and have no buffers for it. It is purely an optimisation to eliminate an unnecessary copy.

It is always possible to queue more requests after proxies and joins. However, joining results in the application being unable to write more requests to the target of the join until after the join is finished, as ownership is temporarily transferred. This would limit how much IPC could be queued. However, there is a simple fix: combine a proxy and a join. Figure 6.4 shows a set up where NGINX needs to send a header, then a file, and then a footer over a TCP socket. This is similar to what sendfile can achieve on FreeBSD. There are two requesters that need to be joined: a pull requester (A) that represents a file, and a push requester (B) that represents the TCP socket. Making a naive join request in (A) pointing at (B) would block (B) from having further requests made until the join finishes. Instead, we create an entirely new proxy socket. Call the request side of this new socket (C), and the fulfil side (D). We insert a join request from (A) to (C), and a proxy request from (B) to (D). (A) will now form requests that represent file data in our new socket because of the join, and (B) will fulfil them via the proxy. Both (A) and (B) can still have more requests queued up as neither were a target of a join or proxy. NGINX uses an indirect request for the header, then a proxy to the proxy socket that will contain file

data, then an indirect request to a footer. We are free to queue more requests in both the File and TCP requester. For example, we may wish to queue another file send from a different file on the same TCP socket while the first is completing.

Once IPC has been queued, the application can sleep while the transfer takes place. It does not need to wake again, regardless of how much data is being sent. In fact, even operations such as closes can be queued. Sockets can be closed as far as the application is concerned before going to sleep and allowing the IPC to take place. The file in this example is probably in turn a proxy target, as shown in figure 6.3. By chaining together multiple sockets in shared memory, CheriOS achieves IPC across multiple compartments, without context switching after the initial set up, and without copying either data *or* requests.

The UNIX abstraction layer in CheriOS implements a sendfile wrapper using a combination of proxies, joins, and manual copying, depending on socket types. The NGINX port selectively uses CheriOS sockets directly to get the best performance for important routines, and applies the trick of creating extra sockets to avoid any blocking as just described. Unlike on other OSs, NGINX on CheriOS can perform multiple sendfiles on the same connection, interleaved with other operations. FreeBSD must wait for a sendfile to complete on a socket before another can be queued, and also blocks behind writing to files normally.

Restricting access

Both the request and fulfil sides of sockets are handled by the socket library. The socket library always runs in maximum distrust mode: both as a caller and callee. Applications can choose to trust the socket library. All the shared memory structures are sealed, and can only be modified by the socket library. Buffers are visible to applications, although the socket library strips permissions to ensure data only flows in the intended direction. Even so, care should be taken in sending mutable structures, as once a capability is sent, there is no taking it back beyond revocation. Temporal safety means that programmers get no nasty surprises due to re-use of memory they send over a socket. Using the distrusting ABI, applications will release only the capabilities that are explicit arguments, so the socket library will be unable to send any other application data. This allows applications to control what data is sent, but not limit where it is sent.

Systems software has many layers, both out of practicality and, in CheriOS's case, in order to increase compartmentalisation. This means the compartment we send data to may not in fact be the intended recipient, but instead may be an abstraction layer. Consider, for example, a filesystem. We may send read and write requests to the filesystem, but the data should only be needed when being read/written to disk. The filesystem should just be using proxies, but we have no way to guarantee that it does not spy on data as it goes past. Conventionally, encryption would be used to protect data from untrusted intermediaries. This is still necessary if data were to leave the machine, but within the machine is unnecessary cost on CheriOS.

Two ways of protecting data have already been described. Sealing data is a CHERI primitive, and foundation signing/locking data is a nanokernel primitive (section 5.3), and both could be

used to make sure data ends up in the right hands. Rather than have applications manually use them, if trusted by applications to do so, the socket layer can enforce restricted access on their behalf. Requesters can *restrict* a socket, which specifies what endpoints may see data. They can do this by providing either a sealing capability, called *sealing restriction*, or a foundation ID, called *ID restriction*. If a sealing capability is provided, the socket layer will seal capabilities to buffers before passing them to a fulfiller's callback function. If a foundation ID is provided, the socket library will require that the fulfiller sign their callback functions with the relevant authority token. Recall how signing works in section 5.3. Although signing is more expensive than sealing (which is a single instruction), and in fact requires an allocation, as long as the callback functions remain the same, the fulfiller need only sign them once. Verifying a signature requires very few instructions, and is done once when fulfil is called, not for every buffer in the data steam.

Both these mechanisms are used in the case study, to allow sending of data over Ethernet without the filesystem or network stack being able to see the data being sent.

6.2 Untrusted drivers

Drivers are typically a mainstay source of vulnerability in OSs: in the Linux/BSD kernel it was found they have an error rate of up to seven times that of other software.[25] Searching the CVE database for 'driver' gives 2215 results.[31] Drivers that run in low rings are afforded high privilege, and their failure can lead to exploitation or a crash of the entire system.

CheriOS not only attempts to de-privilege drivers by putting them outside the microkernel, like other microkernel designs, but also attempts to treat them with the same mutual distrust as any other system component. Even so, CheriOS retains the possibility for zero-copy. Currently, CheriOS features drivers for a number of Ethernet, block, and UART devices.

CheriOS device drivers

CheriOS drivers are user programs like any other, although they will probably hold some different capabilities to allow them access to memory-mapped control registers. They run with no ambient authority, cannot cause corruption to other parts of the OS, and can only access application data if capabilities are explicitly released by the application.

Although all architectural exceptions are first handled by the nanokernel, device interrupts are propagated directly to the microkernel by switching to a pre-registered exception context; see section 5.1. Subsequently, transitioning to a driver is done like any other application. Interrupts are delivered by the microkernel as messages that appear to come from a special interrupt-source activation. A microkernel capability[1] can be used to register to receive messages with a specified activation, and also specify callback arguments to be delivered with the message, which can

[1]In the current implementation there is a single capability the microkernel offers which is the right to register for any IRQ, although this could more sensibly be a capability for each IRQ.

include capabilities. Every time a driver receives a message for a specific IRQ, they must notify the kernel they are ready to receive another. Drivers normally seal capabilities to internal structures they manage, and specify these as callback arguments. This stops any driver state being visible to the microkernel. Messages can also arrive from applications, and so most drivers will have a simple event loop that sleeps on their message queue and takes an appropriate action as messages arrive, whether that be interaction from an application or an interrupt from the device.

The microkernel's message send is generally only used for handshaking and event notification; the more fully featured CheriOS sockets described in section 6.1 are used when a high-bandwidth data stream is required. CheriOS sockets allow buffers to be safely shared between drivers and processes (with flexibility on which allocated the buffer). CheriOS drivers that are DMA capable translate between the scatter gather descriptions from the sockets they manage into their device specific format.

Typically, devices have a number of control registers that are mapped into the address space and so can be controlled using normal load and store instructions. Drivers require access to memory-mapped control registers that other programs should not. Drivers can be granted the capability to control specific devices by releasing a bounded CHERI capability to the memory-mapped control region for that device. CHERI capabilities are fine-grained enough to grant the rights to control a specific device. The nanokernel is aware which physical addresses are IO addresses, and which are RAM, and this information is embedded in ROM. It knows nothing about which devices occupy which regions, as it is still up to the OS to correctly decide which driver should get which MMIO region. It is a hard-requirement for security that the nanokernel know which regions of memory have RAM semantics (any other regions are IO memory). This is possible in our system as the locations that can contain RAM are statically configured. This may not be as possible on other systems, in which case a trusted boot would be required to provide the nanokernel with this information when it is loaded.

MIPS has windows to the physical address space embedded in its virtual address space. It is therefore possible for a driver to be granted a 'physical address' capability that refers directly to memory-mapped registers, generally uncached, and without the ability to store capabilities. How these are provisioned from the nanokernel is discussed in section 5.2. However, in most cases, drivers are instead provided memory capabilities to a software-controlled TLB-managed portion of the virtual address space. Capabilities to virtual addresses are acquired using reservations, as discussed in section 5.2. Providing access to devices via reservations delivers a number of key benefits. The temporal safety property of reservations means that multiple drivers cannot accidentally govern the same device. Furthermore, reservations are revocable. If a driver misbehaves, the reservation can be revoked, the driver restarted, and a new reservation granted to the new instance of the driver.

DMA and IOMMUs

Software copying data can significantly impact the performance of IO. In order to speed up operation, some devices are capable of DMA (direct memory access), which allows them to

directly load from, and store to, system memory. Drivers can write descriptors of operations (such as copying a block from a block device into RAM) that need performing to main memory. These descriptors can be parsed by devices themselves, and the operation undertaken without disturbing the CPU's normal operation, using interrupts to communicate when transfers have finished.

Absent of DMA, CHERI provides all the power we need to create secure device drivers. DMA, however, poses significant problems. Devices effectively have a capability to access all of memory, and any driver which can control a device can subsume its capabilities. Removing DMA from a system would not be acceptable for performance reasons, and so a way of constraining it is required.

One technology that already exists in the space of DMA protection is the IOMMU. IOMMUs offer the virtual memory facilities that MMUs do, but for DMA rather than CPU memory access. This is especially useful as devices need not know this happening, and so need no modification. IOMMUs not only allow automatic translation, which is a useful facility, but also a rudimentary form of access control. A separate IO space can be constructed that allows access to only some parts of memory, with either read or write privileges. Although this gets us some of the way, IOMMUs are very coarse-grained and do not support CHERI-like capabilities.

Although IOMMUs could feasibly provide a secure access-control mechanism, their current uses are subject to both spatial and temporal safety issues. Dedicating an entire page to an object is often considered undesirable, and so OSs will use the rest of the page for other data, such as OS-private metadata. Devices are free to read and write the data in the page around the intended buffer, and so can modify nearby kernel structures to achieve arbitrary code execution. Memory can also re-used by the OS before IO-MMU windows have been fully closed, also leading to memory corruption attacks. Work in this area has shown that most major OSs (OSX, Linux, FreeBSD) are still vulnerable to malicious devices, even in the presence of an IOMMU.[86] Rather than rely on IOMMUs for access control, we might instead use them the same way CheriOS uses the MMU, having a single IOMMU address-space and then using capability-based access control. See appendix D for an outline of such a solution.

Another solution would be for devices to participate with CHERI and demand CHERI capabilities be used in their descriptor structures. This would solve the problem outright; applications could pass capabilities to their buffers to drivers, which could in turn put them in descriptor rings. Sadly, this solution is not immediately tractable as devices would need modification, and there would be challenges to resolve with respect to revocation and address-space management. Furthermore, it may never be the case that we trust devices. Rogue or buggy hardware/firmware is commonplace.

6.3 Case study: CheriOS stack with NGINX

To illustrate uses of all the facilities offered by CheriOS, we will look in depth at how the webserver, NGINX, makes use of them. NGINX was ported to mostly make use of CheriOS's

UNIX abstractions for files, but some IO routines we were rewritten for performance reasons, and a CheriOS-specific abstraction layer was written for OS interface not related to files. The NGINX port for CheriOS weighs in at 1348 lines of C, including headers, but not including comments and blank lines.

System setup

The network stack on CheriOS is a port of LWIP (LightWeight Internet Protocol). Only minor additions were made to glue LWIP to CheriOS sockets, as well as to ensure CHERI primitives were properly used for bounds checking. The NGINX port is set up (on CheriOS) to use a polled event model, and sendfile for file transfer. Custom wrappers for most OS related calls were written to get NGINX working on CheriOS.

The Filesystem in use is an open source port of a 32-bit FAT implementation (chosen for the sake of simplicity), modified to use proxying rather then copying in to and out of its own buffers. It is sadly quite basic and supports no concurrency. Due to the nature of FAT, it is very bad at batching operations due to the linked list structure of FAT metadata. Each read of the FAT requires a round trip to complete before the next read can be queued. This means that although CheriOS supports asynchronous IO, these reads tend to form a synchronous bottleneck in some cases. The block cache and drivers are custom written for CheriOS, make direct use of CheriOS primitives, and can handle both batched and asynchronous IO.

Both NGINX and LWIP use custom allocators. NGINX's never reuses memory and uses malloc for allocating buffers for its own pools, and so will be temporally safe. LWIP has some type-stable static object pools which it might be able to use in a temporally unsafe way. Crucially, receive buffers in LWIP, which get passed up to applications, are allocated using malloc and so are temporally safe. The prevalence of custom allocators in high performance code highlights how, even with good provision of temporally safe memory, programmers still introduce potential sources of vulnerability. All custom-written programs and drivers use malloc for allocation, so are guaranteed to be safe.

We compare CheriOS's implementation to NGINX running on CheriBSD (a port of FreeBSD that runs on CHERI-MIPS). The networking stack in FreeBSD, as well as the filesystems and drivers, are more fully featured and developed than those of CheriOS, but still offer the best practical comparison to the state of the art. In order to keep the comparison fair, we also enable sendfile for NGINX on CheriBSD, but allow it to use kqueues rather than the more primitive polled model. CheriBSD has no compartmentalisation within the kernel, and although it runs on CHERI, it does not offer all the guarantees that CheriOS does. Namely, CheriBSD does not offer any temporal safety, runs without memory safety in the kernel, and does not support mutual distrust or attestation.

Compartmentalisation

In order to minimise the potential damage of any exploits leveraged against CheriOS, we attempt to break down OS services into multiple compartments. In the webserver setup, the important compartments are the:

- Nanokernel
- Microkernel
- Socket Library
- Block Driver
- Block Cache
- Filesystem
- NGINX
- LWIP
- Ethernet Driver

The Block Driver, Block Cache, Filesystem, NGINX, and LWIP networking stack are all separate processes, each in its own compartment. The Ethernet driver is linked with the LWIP application, but is still its own compartment. The nanokernel, microkernel, and socket library compartment are used by all the others as dynamic libraries. Each application has its own event loop, uses OS-facilitated IPC for handshaking and control messages, and uses CheriOS sockets for large-scale data transfer.

To see how most of the important compartments fit together with respect to communication, see figure 6.1 again. Although not shown, the socket library compartment covers the entire socket graph, and the other compartments have to call into it to make modifications to the shared structure. Communication with the microkernel is done as context switches and message sends occur, and and all compartments frequently interface with the nanokernel to use objects like reservations and foundations.

As illustrated in the other chapters, CheriOS is configurable with respect to security, and configuration choices made have an impact on performance. In order to emphasise that good security need not come at the expense of good performance, we keep the majority of the features on. Namely, the nanokernel, microkernel, socket library, block cache, NGINX and Ethernet driver are run in a fully distrusting mode as they handle sensitive data by design. The filesystem and LWIP stack can be more trusting as they handle no sensitive data due to compartmentalisation (see section 6.3). The compartments run in a distrusting mode are all loaded via foundations (so the program loader cannot influence them), have temporally safe stacks enabled, and use the calling convention that ensures cross-compartment CFI and restricts capability flow to explicit

arguments only. Features that are not configurable, because they are fundamental parts of CheriOS's design (like temporally safe heap memory, bounded functions and objects etc.) are ubiquitously on in all compartments. Some exceptions to the above are the nanokernel, which has no stack to be considered temporally safe, and the microkernel, which does not perform dynamic allocation and does not make use of temporally safe stacks.

The result of this compartmentalisation is that the data from the HTTP transfer can only be accessed by the block driver/cache, NGINX, the Ethernet driver, the socket library, and the nanokernel (although the latter two make no attempt to do so). Internal state of distrusting compartments is completely protected from others, apart from the nanokernel which can inspect the state of any compartment.

Least privilege data access

Least privilege dictates that we should only give compartments the least privilege required to perform their task. The data being served by NGINX travels from a storage device and then out 'on the wire', and is in some way handled by most compartments listed in the setup. However, the *contents* of the data being sent are not really required by most of the compartments. We will use a combination of CheriOS primitives to control which compartments get access to data, trying not to impose too much overhead, while also not breaking any layering. Read about reservations in section 5.2, foundation locking and signing in section 5.3, and socket restriction in section 6.1.

Socket restriction is the main technique used for restricting access to stream contents while still allowing control of the stream itself. The LWIP stack should not need to see the contents of data being sent[2]. However, the Ethernet driver will need to see the contents. The Ethernet driver uses type reservations to acquire a unique sealing/unsealing capability. It then subsets it to create a sealing-only capability, and signs this capability using its authority token. It advertises this signed capability to the OS.

When NGINX opens a TCP socket, it asks the OS (currently the namespace service for lack of better choice) for this signed sealing capability. It checks the signature against the ID of the Ethernet driver it trusts, and then performs a sealing restriction on the TCP socket with this sealing capability. The result of this is that any capabilities received by LWIP from the TCP socket will be sealed, and un-sealable only by the Ethernet Driver.

The Ethernet Driver offers a function to the network stack that will take a sealed capability and checksum its contents, returning a sealed capability to a buffer with the checksum. This is provided as some devices do not support offloaded checksum calculation. LWIP can use this function to blindly checksum data, and then include the checksum as a part of the packet, seeing neither the data, nor the checksum (which would reveal information).

Obviously, this example still sends data in the clear on the link-layer, and is no substitute for setting up a cryptographically protected end-to-end secure channel. The intent of enforcing

[2]Checksums are evaluated by calling into a specialised compartment, which shares authority with the Ethernet driver. The returned checksum is protected in the same way as data, and so does not reveal the data either.

least privilege with respect to data access is to provide defence in depth. If TLS were in use, which currently CheriOS does not support, we could enforce that the TLS compartment was uniquely able to act as an endpoint in the same way we did for the Ethernet Driver, which allows ”end-to-end” protection to extend to the user application, not just a TLS implementation.

Socket restriction is also used on the files NGINX opens. NGINX restricts files using ID restriction, specifying the ID for the block cache. This means that the filesystem cannot read and write its end of the file socket directly, it can only proxy to the block cache.

In order for this example to function, NGINX needs to know the foundation ID for some OS components. It is not suggested that foundation IDs be hard coded into NGINX. Instead, the foundation ID of a mediation service should be embedded in NGINX. The mediation service should ask the OS for the foundation IDs of its services, decide if they are valid, sign the IDs using its own authority token, and subsequently return the signed IDs to NGINX. Deciding which services are valid is a policy problem, and one quite similar to validation of dynamic libraries, a space where solutions already exist. Possible workable policies are: accepting binaries with cryptographic signatures provided by vendors, or using an online lookup step to query an up-to-date list of safe services. Currently, CheriOS has no such indirect service, and we just build NGINX with knowledge of the most up to date foundation IDs it should trust. This requires recompilation of NGINX every time a system service is updated, which is passable for testing purposes.

File encryption The filesystem allows programs to specify an AES key when opening a file, and data will then be encrypted in CBC mode when read or written. NGINX could employ this to protect against not only physical attacks on storage devices, but against parts of the OS such as the block-device driver and the filesystem from being compromised. The best place to perform encryption and decryption is where data is already cached, the block cache.

The first requirement is for NGINX to provide an AES key to the block cache. However, not only should applications not be interacting with the block cache directly, the block cache has no notion of files. It is the filesystem’s role to tell the buffer cache which key should be used for which block. It does so with an out of band socket request which allows the filesystem to interleave IO operations on different files. According to least-privilege, the filesystem should only be able to reference the key, not see its contents. To this end, an asymmetric foundation lock is used. NGINX locks a structure containing the key using the block cache’s foundation ID, and this locked key is given to the filesystem. The structure also describes on which requesters the key can be used, so the filesystem does not mismatch keys.

Combined with socket restriction, this means only the Buffer Cache (which performs the decryption), NGINX (which owns the data), and the Ethernet Driver (which sends the data) can ever see plaintext. Other attacks, such as traffic analysis, are still in scope with these changes.

There is still the question of how keys should be provisioned by NGINX. NGINX could receive its keys when it starts up using an online step. Another option is to use the nanokernel,

Activation	Time	Switches	Sent	Received
NGINX	34%	3.09	1.08	1
LWIP	29%	2.06	1.04	1.05
FatFS	17%	7.01	3	1
Block Cache	11%	4.01	0	3
Block Driver	4%	0.01	0	0
Other	5%	1.2	0	0.1
Kernel	0%[4]	0	1.05	0
Total	100	17.38	6.17	6.15

Table 6.1: *Per activation statistic breakdown for an average request.*

which will convert an authority token into a cryptographic key, based on both an internal secret and the corresponding foundation ID to the authority token. This key will be the same each time a foundation with the same ID is loaded, and can be used to encrypt the files it writes to disk, or more practically, to decrypt another key that stays constant over multiple versions of NGINX. CheriOS has a working demonstration of directly using the cryptographic key derived from an authority token to read and write files.

File encryption was not turned on for benchmarking, meaning the block driver can still see the plaintext data. However, socket restriction is in use, and so neither LWIP nor the FAT filesystem[3] can read data flowing through sockets. This choice was made as benchmarking a non-production, third-party AES routine used for prototyping was not an interesting comparison point. The implementation is almost certainly not secure, and contemporary hardware can offload a lot of AES work that CHERI-MIPS cannot.

Performance

CheriOS workload breakdown

To aid better understanding of how CheriOS handles a HTTP request, we give here a breakdown of scheduler-activation activity, and trace the exact execution of the system when handling a 1KB HTTP request. Recall an activation is not entirely the same as a compartment. A single thread of execution can move through several compartments, and the benchmarks routinely do: for example, all threads will likely move through the socket and microkernel compartments. The LWIP activation also handles both the LWIP network stack and the Ethernet driver, which is in a different compartment again.

[3]The filesystem can still read arbitrary metadata blocks, although it does not do this for file contents. File contents never enter the filesystem compartment, even though they could if a logical error occurred and the wrong block were read. Filesystems, such as FAT, have a simple static split between metadata and file data, so one could envision enforcing the filesystem never gets access to file data.

[4]Time spent in the interrupt handler is accounted to the interrupted activation.

Table 6.1 shows a breakdown of cycles spent, context switches, and messages sent and received for the average HTTP request. The 'Other' row accounts for other tasks that were running, mostly consisting of time spent idle (the workload is CPU-bound so there is mostly no idle time), allocating virtual memory, and zeroing physical memory. The 'Kernel' activation is a dummy activation bound to the exception context. It generates messages from interrupts and also switches based on timer interrupts. The fractions are a result of infrequent events, such as timer interrupts and obtaining more temporally-safe memory. Otherwise, there are quite reliably 17 switches and 6 messages across the entire system, per request. Using section 5.4, we can estimate how many cycles this should cost. It is somewhat difficult as some messages are synchronous and others not, some calls distrusting, and some context switches without a message enqueue. However, an untrusting message send will entail 2 context switches and a message sent, so 8 of those (12774 cycles) is probably a good estimate. This is ∼3% of the total time spent, which for micro-kernel design with a system-heavy workload is acceptable. Looking even closer, to see just how infrequently CheriOS context switches, here is a trace of the path for a request. Each line represents a context switch; it is nearly always followed exactly:

1. A TCP SYN packet arrives, generating an interrupt. The microkernel sends a message to LWIP.
2. LWIP wakes up due to having a message. It then enters polled mode to avoid interrupt storms. It performs a TCP handshake and sends an asynchronous message to NGINX with a socket half. The HTTP request arrives before LWIP sleeps, is seen by polling, and LWIP puts data in the socket that it already sent to NGINX. LWIP sleeps.
3. NGINX wakes up as it receives its message from LWIP with the socket half. It creates its own socket half, and puts it in shared memory for LWIP[5]. It then sees there is already data in the socket and reads the HTTP request. This causes it to open the file by sending a synchronous message to FatFS.
4. FatFS is switched to. It sends a synchronous message to the Block Cache to read the FAT.
5. The Block Cache is switched to. It reads a block and returns to FatFS.
6. FatFS performs a second block read of the FAT and sends a second synchronous message again.
7. The Block Cache once again returns the block to FatFS.
8. FatFS has now opened the file, it creates a socket half and returns it to NGINX to represent the open file.
9. NGINX receives the message response. It performs a filesize request on the socket and waits for the response[6].

[5]This socket half is actually for a *future* connection, not the current one. Socket halves are buffered at the same depth as TCP backlog to avoid having extra switches. This buffer is exchanged and filled when listen is called.

[6]The filesize is required for the file length in the content header of the HTTP response.

10. FatFS wakes due to having a request in a socket, it handles the filesize request and goes back to sleep.

11. NGINX wakes up. It formats the HTTP header. It sends the header using an indirect request, and the file with a join request on the file socket, and a proxy request on the TCP socket. It also queues a write to the log file. It then queues closing both the file and the TCP socket, as keep-alive was not specified. NGINX has no work to do, so it sleeps waiting for the next connection.

12. FatFS wakes up as there is a request to read the file. It sends a synchronous message to FatFS to read the FAT to work out which blocks this corresponds to.

13. The Block Cache responds to the synchronous message.

14. FatFS now knows which blocks need to be read, it sends a number of seek and proxy requests to the socket it has open with the block cache to handle the file read. It goes to sleep.

15. The Block Cache wakes up as there is a read on a socket. It handles the seeks, proxies, and joins and then provides capabilities to its buffers as a response.

16. LWIP wakes up as there is a write on its socket. It handles writing the HTTP header, then the data proxied/joined by the Block Cache, then sees the connection should be closed and closes it. It goes back into interrupt mode as it does not expect any more immanent packets.

17. FatFS wakes up as the proxy has finished. It sees the close NGINX queued and closes the file.

If the reader is wondering when NGINX cleans up its socket halves, it will do so lazily when future file operations are performed. Calls to close a file will queue the asynchronous close, and also free resources for previously closed files that have finished closing. The same is true for the TCP connection. The details for how LWIP enables and disables interrupts are not so important; the general idea is it keeps them off while scheduled and switches into polled mode when all active connections have outstanding data being handled. Log file writes are also handled lazily by FatFS whenever it is scheduled. This is due to a socket option used when opening the log file.

Note that out of the 6 message sends, 3 were due to needing to serially read the FAT. FAT does not lend itself well to asynchronicity; it is at heart a linked list. A tree-like filesystem would be able to queue far more block-level reads between switches. LWIP performs particularly well at minimising scheduling slots, being switched to first to open the connection and see the HTTP request, and then a second time to send the result and close the connection. There is no obvious way that this could be any better. NGINX is also pretty good, combining most of its operations in one batch. There is no reason to suggest that the 3 switches could not be reduced to 1. Requests can be placed in a socket half before it has been sent and so there is no need to make open synchronous, as it currently is. Asynchronous creation of sockets already happens with TCP listen. Batching the filesize with the sendfile would also be possible using CheriOS

sockets; filesize is just an out of band request. NGINX would have to make the filesize request (which takes a pointer of where to write the filesize) point to the length field being sent in the header. Formatting requests would also need to be built into the socket layer to send the size as a string. The effort did not seem worth it for this benchmark, but would be interesting future work.

An interesting question arises as to why LWIP does not ever wake in-between NGINX writing the HTTP response header (step 11) and the Block Cache proxying the data from the file (step 15). This is not luck. Look again at figure 6.4 for a depiction of the step in discussion. NGINX knows that there is more data to come after the header, and that it is probably not worth waking up the networking stack if only the header is available to send. Instead, it takes the notification token from the TCP socket and transfers it to the target of the join from the file socket. This means that only when the first part of the join request is fulfilled (which is when file data is first available to the TCP socket) will a notify be sent to LWIP. This does not mean that LWIP cannot see the header data in the socket. If it is scheduled for some other reason, such as a timer interrupt, it will notice that there is some data in the socket to be sent, even though it was not notified. This results in a trade-off wherein LWIP will not wake up to send a small amount of data, but if it does, it will send as much as possible before going back to sleep. Owners of a socket are always allowed to choose to not notify when they make or handle a request, and can also extract and do as they please with the notification token to notify the waiter on a custom condition.

Transferring larger files follows practically the same path, only differing at the end where it will switch back and forth between the block cache and LWIP if the request queue is not large enough to hold references to the entire file.

For a steady state workload, if only a single core were available, then the same chain of switches would be made, but more work would be done in individual compartments to completely drain / fill work queues before making any switches.

In a steady state transfer, where there are multiple cores, the steps above become pipelined and parallel. Individual compartments would work through incoming requests and translate them into outgoing requests, only sleeping if incoming queues were to empty, or outgoing queues were to fill. Concurrent access to the queues is handled internally by the socket library. For example, steps 1, 2, and 11 would be alternated between by a thread in LWIP, without sleeping, until there were no more packets to receive or send.

Compared to CheriBSD

In an attempt to make a fair comparison, NGINX was configured to take advantage of sendfile on CheriBSD, and was also compiled to make use of capabilities for pointers in order to provide spatial memory protection. However, the CheriBSD kernel itself is still a ‘hybrid’, able to use capabilities explicitly (in order to serve the capability-fied userspace) but otherwise using only a single default capability for the entire memory space internally. Although capabilities are enabled for NGINX, most of the benefits of CheriOS are not present in CheriBSD. For

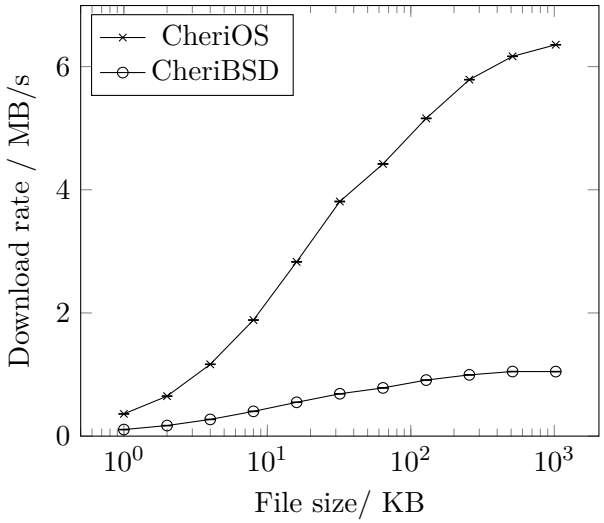


Figure 6.5: *Mean HTTP transfer rates using NGINX on CheriOS and FreeBSD, as file size scales.*
Error bars show one standard error (n=1000).

example, there is no mutual distrust or compartmentalisation offered beyond what FreeBSD would normally offer. CheriBSD does offer spatial safety for most objects in userspace. NGINX has been finely tuned to work on top of FreeBSD, and by extension CheriBSD. However, it is worth noting that the hardware in use is of significantly lower performance than would be expected, and so CheriBSD is not necessarily optimised for such a device.

Both CheriOS and CheriBSD drivers support DMA, however DMA was non-functional (on either CheriOS of CheriBSD) at the time of writing due to FPGA limitations. Both CheriOS and CheriBSD were designed with DMA in mind; CheriOS specifically tries to leave scatter/gather operations to DMA at the ends of IPC chains. Both OSs should take roughly equal penalties for not having DMA, but results may differ from what is presented if DMA were supported. The NGINX port for CheriOS knows how to use CheriOS-specific IPC features in a few places, for example: asynchronous logging, closing, and sendfile. Both CheriBSD and CheriOS were run in a benchmarking mode with debugging features and asserts disabled, and any fastpaths enabled.

Each test was comprised of 1000 requests to NGINX. As with all other benchmarks, we run on an FPGA, with a client on a separate machine with a direct Ethernet connection, and each request brings up and tears down a TCP connection, as well as opening and closing the file being served. It is very likely that both OSs will cache the file, and we run several requests on both, in advance of the test, to ensure caches are warm. CheriOS caches low-level blocks only, while CheriBSD's unified buffer cache can cache both files and device blocks. That is, CheriOS pays the cost of walking FAT metadata every open and read, while CheriBSD uses its vnode caches to accelerate these processes.

Figure 6.5 shows the average download bandwidth achieved as file size is varied (doubling each interval) between 1KB and 1MB in size. CheriOS does considerably better at all points on the graph, being about ∼3.5x faster for smaller requests (where the overhead of opening and

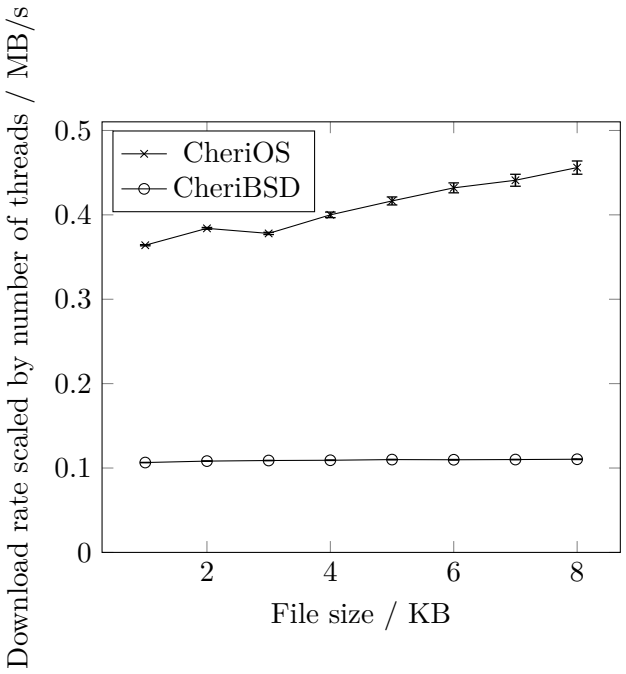


Figure 6.6: *Mean HTTP transfer rates using NGINX on CheriOS and CheriBSD, as number of connections scales.*
Error bars show one standard error (n=1000).

closing sockets dominates), and up to 6x for larger files when raw network IO speed dominates. CheriOS's asynchronous IPC infrastructure was designed with multicore in mind; however, as of the time of writing, multicore CHERI-MIPS was not mature enough to run CheriOS. Even so, in order to illustrate that neither OS had trouble supporting concurrent requests, figure 6.6 shows what happens as concurrent requests are scaled. The rate given is *not* the average download rate across all threads, it is the average download rate for a single thread (which is understandably lower under concurrent load), scaled by number of threads. Both systems do better when given more work to do at once. However, CheriOS sees a larger increase (25% compared to 4% at 8 threads), giving it advantages under concurrent load. This is understandable as CheriOS can batch arbitrary IO requests.

Understandably, more mature OSs like CheriBSD are slowed down due to supporting more features, even when off. Layers of abstraction tend to not give better performance. However, it is important to know how much of CheriOS's performance comes down to design choices, and how much to simplicity. To this end, a more in-depth analysis of a single data point was performed: the 1KB, single-concurrent-client HTTP request. The CHERI-MIPS FPGA implementation has in-hardware stat counters for a number of important metrics. CheriBSD can collect these globally, whereas CheriOS virtualises the counter across activations. In order to turn on tracking in CheriOS, the message send fast path needs to be disabled. Combined with the cost of virtualisation of these counters, CheriOS takes a significant (11%) performance hit just for tracking them. No changes were made to CheriBSD as the performance-tracking library in use did not support fine-grained tracking. Even with this performance hit for CheriOS, there is still a 3x difference

between the two OSs that needs explaining. First we will look at micro-architectural indicators of where the difference may come from.

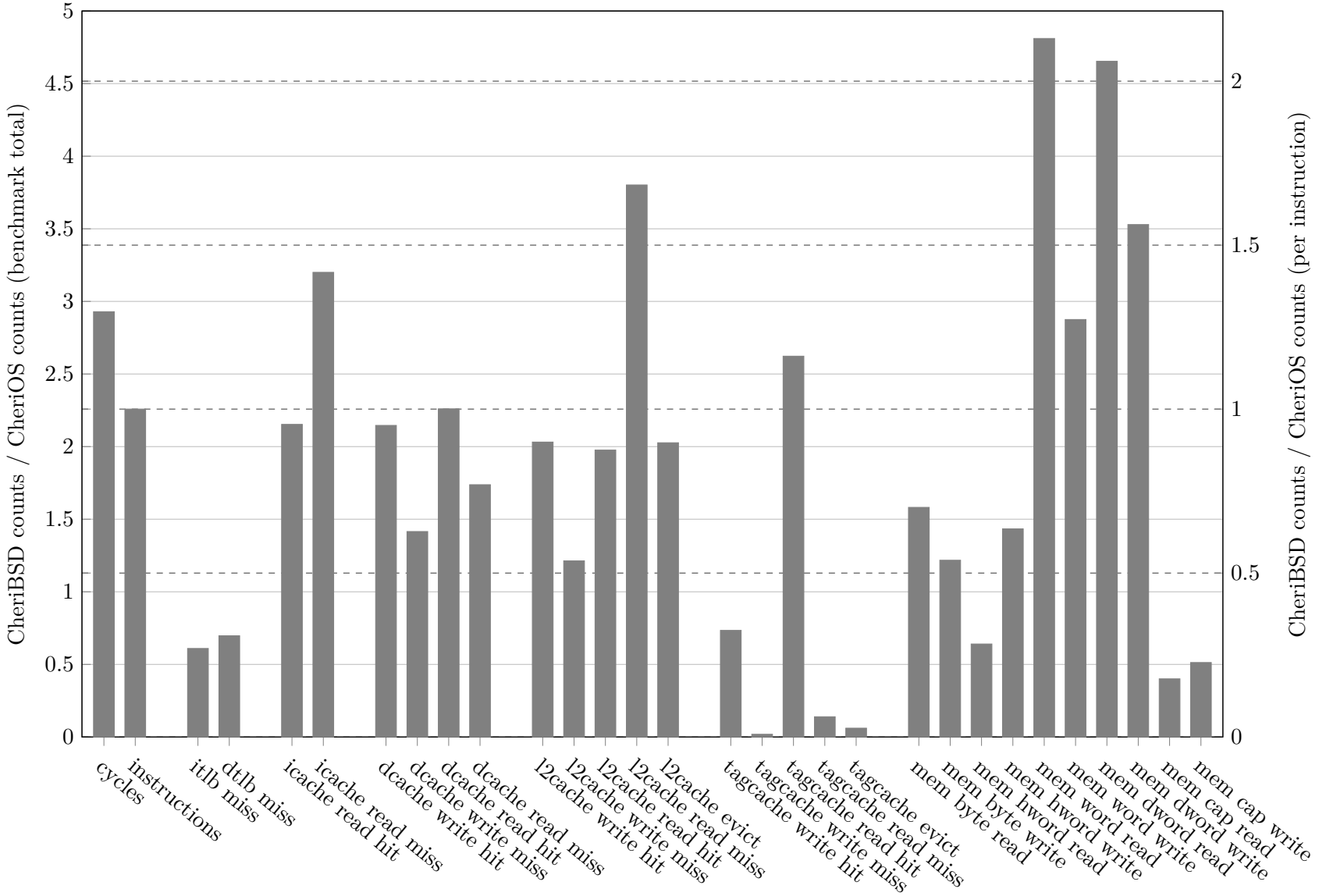


Figure 6.7: *Microarchitectural counters for CheriBSD divided by CheriOS for a similar workload. A bar greater than 1 indicates CheriBSD has a higher count.*

The 1KB file was requested 10 000 times for the purposes of getting reliable counts. Figure 6.7 shows the system-wide counters for CheriBSD relative to CheriOS. A value of 1 means CheriBSD had as many of an event as CheriOS, greater than 1 means CheriBSD had more. The left axis shows the relative counts for the entire benchmark. The right axis shows the relative count normalised for number of instructions executed on the respective system. For example, using the right axis for the cycles count allows comparison of CPI.

Looking first at the normalised version, a striking value is the number of cycles. CheriBSD needs about 1.3x times as many cycles to execute the same number of instructions as CheriOS. Probably to blame is CheriBSD's cache performance, due to both a larger dynamic footprint and locality of access. In both the L2 and I-cache, CheriBSD both hits less and misses more. On the L2 cache there are also more accesses per instruction as a total. This likely results in much of the difference in CPI.

The only area where CheriOS seems significantly worse than CheriBSD is with respect to TLB misses and tag cache behaviour. More TLB misses are not surprising; CheriOS uses virtual memory pretty much ubiquitously, even for OS components. CheriBSD, on the other hand, uses physical addressing in the kernel, where most of the work of this benchmark is done.

The CheriBSD kernel makes less use of capabilities as compared to CheriOS, explaining lower use of the tag cache (compare the number of capability memory operations). Recall on CheriBSD only userspace was compiled to use capabilities, so if a more userspace-heavy benchmark were run, the figures may look more equal. It also seems that, overall, the cost to CPI of TLB and tag cache misses is nowhere near as bad as cache misses in the instruction cache and L2.

CPI goes some way to explaining why CheriOS outperforms CheriBSD. However, there are still significantly more instructions executed on CheriBSD relative to CheriOS. One cause is memory operations, see figure 6.8. CheriBSD moves more data in total, but there is an even larger overhead in the number of memory instructions executed, as smaller units are used. CheriOS can take advantage of a single address space to reduce memory operations. On CheriOS, when NGINX reads a HTTP request, it is reading directly from the buffers where raw packets arrived[7]. This is just not possible to achieve on existing OSs without either fine-grained security, or abandoning a pretence of security altogether.

There is also the high cost of system calls. A CheriBSD system call is about as expensive as a CheriOS context switch, and even compared to a distrusting CheriOS call this is quite a high cost. CheriBSD will also make many more of these than CheriOS. For example, CheriOS can perform a batched sendfile, log write, and close on both a socket and file before sleeping again in poll. CheriBSD will have to resume NGINX after each of these steps to continue where execution left off. Switching between the kernel and userspace repeatedly not only has a high cost in instructions executed, but also spoils temporal locality. This may contribute to the number of cache misses that CheriBSD suffers. As an experiment, to see the cost of one of these calls, logging was disabled on NGINX on CheriBSD. This saves only a single write system call for a

[7]NGINX chooses to later serialise the data into one of its own buffers, but there is no reason a more CheriOS-aware port could not parse the data directly.

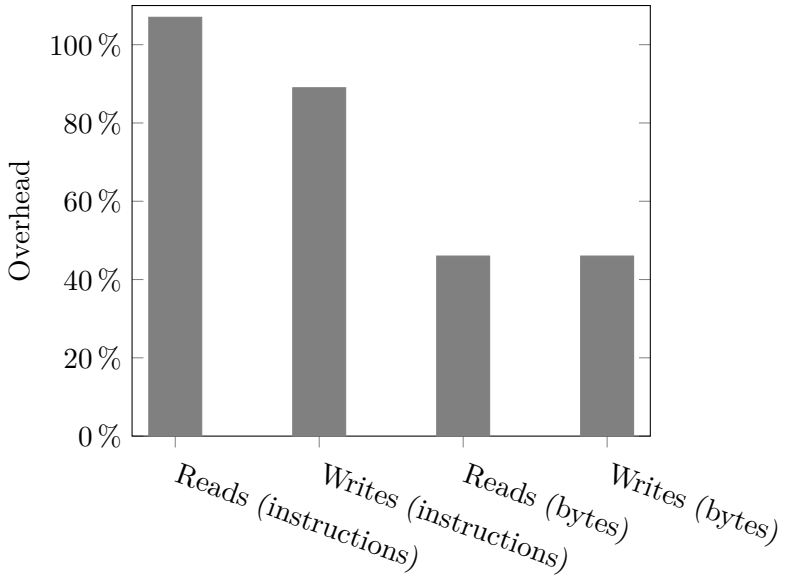


Figure 6.8: *CheriBSD R/W Overhead relative to CheriOS baseline running HTTP benchmark*

line of text on each request. Even so, it results in a 6% performance improvement for the entire request.

Some of the other benefits of batching IO operations are also less obvious. For example, CheriOS sends one fewer TCP packet than CheriBSD for the same request. It does so by piggybacking the FIN packet with the last data packet being sent. Although requests are handled one at a time by the socket callback functions in the TCP stack, multiple requests may go into one packet. If a number of write requests followed by a close request are seen at once, this will result in a packet that contains both data and a FIN. CheriBSD will push the response packet created by sendfile out its networking stack as soon as possible, at the behest of NGINX, in order to achieve a good response time. When a subsequent call to shutdown the socket arrives, it is too late, as the packet has already been sent. No modification was made to NGINX to achieve this behaviour; it is a natural consequence of asynchronicity and batching. We could imagine adding more flags to CheriBSD's syscall interfaces to achieve a similar result, and previous work has done exactly this[95] to improve the performance of sendfile. However, in general, combining syscalls into 'all-in-one' versions (like sendfile is itself) in order to present opportunities for optimisation results in more complicated OS interfaces, and places a burden on application writers to use them. Shared-data-structure based interfaces are better than call-based interfaces at combining operations without a combinatorial explosion of calls, and so win performance without special effort.

CheriOS also has an advantage with how it represents data streams. CheriBSD uses a linked list of 'mbuf's to represent a data stream. This has the advantage of making it easy to re-order and splice data, but means that these nodes need to be allocated and freed dynamically. CheriOS, on the other hand, makes use of very simple ring-buffer based queues. It is much faster to allocate a request in one of these ring buffers (a single addition) than allocate a new mbuf header. The CheriBSD headers are also larger than CheriOS requests. Although arbitrary re-ordering is harder

on CheriOS, proxy and join requests are versatile enough to merge data streams with headers, as is required for protocol encapsulation. This may account for why CheriOS can achieve much higher transfer rates, especially for large files.

Overall, it is very difficult to pinpoint the exact causes of performance changes in a clean-slate OS where multiple design choices have differed. However, the ability to run the same applications on the same hardware gives reasonable confidence in the overall effects, and the benefits of CheriOS-style IO are clear.

6.4 Related work

Here I give an overview of other IO-accelerating methods, both those that attempt security, and those where pure performance was the only consideration.

Copying reduction

IO-Lite IO-Lite[102] tries to reduce IO overhead from unnecessary copying and double buffering by unifying buffering across the system, and only passing around data by reference. All data is stored in read-only buffers, and data is passed using a list of pointer/length pairs to these buffers. ACLs with processes as subjects and pages as objects are used for access control. This has some similarities with how CheriOS sockets represent data in a push socket. IO-Lite is not stream-oriented; the lists themselves are copied by value and copies can be arbitrarily manipulated. Passing the gather list by value means that unnecessary copying is present in a fashion, but with respect to the references rather than the data. These lists can in fact be quite large. It might be suggested that compaction is the answer, but this reverts back to unnecessary copying. CheriOS solves this problem with proxy and join requests, so passing by reference can be used recursively.

IO-Lite also suffers from the restrictions that MMUs bring. Although copying is minimized, shared mappings still need to set up for the buffers being shared. Also, shared mappings cannot handle sub-page objects without sharing the entire page. The paper argues that a page can be re-used for multiple objects as long as the same ACL is used for both objects. However, this will pose an issue if ACLs ever need changing, and assumes that objects are allocated with advance knowledge of the ACLs. This is probably not the case where applications wish to send their already allocated buffers via IPC. Due to using a SAS, and capabilities rather than ACLs for access control, CheriOS does not suffer from these problems.

Xen The challenge to provide low overhead IO between CheriOS compartments is very similar to one faced in Xen.[12] Xen is hypervisor that allows running multiple guest OSs at near bare-metal speeds. Xen splits drivers into a front-end (Guest driver) and back-end (Host driver[8]), and requires a low-overhead, copyless mechanism to transfer data between them. Xen's I/O rings, like

[8] In earlier versions of Xen, the host driver was a part of Xen. In newer versions it is part of a privileged guest OS.

CheriOS sockets, are ring-buffer based single-producer single-consumer request queues[9], although they support an explicit response for each request. They also allow responses to come out of order. Front-end drivers make requests to be consumed by a back-end driver, which will multiplex access and provide a sharing/accounting policy. The format of each request in a Xen IO ring is a scatter/gather operation, similar to that of a CheriOS indirect request. Memory is mapped in both the front-end and back-end drivers' address spaces so that requests can reference shared memory. The references themselves are just integers, not capabilities.

Sharing on the granularity of a page can cause memory-safety issues. This is the same problem faced by IOMMUs discussed in section 6.2. Setting up shared page mappings is also expensive, both up-front for setting up pagetable entries, but also on-going due to TLB pressure. Xen only envisages a single layer of indirection (the back-end), and so does not have any analogues for CheriOS proxy/join requests. CheriOS sockets are intended to not only allow for compartmentalisation of device drivers without copying data, but also to extend the IO mechanism all the way into the application layer. There is no trivial way to extend Xen IO rings into userspace, like we do with CheriOS, without a sacrifice to either security or performance. Because of the coarse granularity of paging as an access control mechanism, user applications would have to allow entire pages to be read, even if only a few bytes were being sent. Even if we were happy to trust back-end drivers to only read the bytes we intend, the cost of setting up a mapping for just a few bytes would be prohibitive. The cost of delegating access with CHERI is much cheaper; it comes for free when sending the pointer as a capability.

CheriOS sockets could easily be adapted to allow responses to individual requests as Xen does. However, it is a deliberate design decision to not allow out of order completion. This allows the socket library to handle the complexities of partial request fulfilling, backlog, and synchronization, while providing a simple source/sink interface. Requiring applications to implement something with the complexity of a device driver just to read a byte stream would be inefficient. One might imagine a more specialised CheriOS socket variant to allow out of order completion (with proxying/joining to normal sockets handled by the socket library), but it would necessarily have a more complex interface.

Spin The Spin[16] OS, like CheriOS, uses co-location of programs and the kernel to achieve improved performance, especially for IO-heavy tasks such as networking. They too make the observation that a lot of overhead can be spared by removing the need to copy between user and kernel space. In Spin, this achieved by extending the kernel via dynamic linking, and controlling access to the interface via software defined capabilities. This is not too dissimilar to how CheriOS dynamically links every userspace program with the kernel, providing hardware-enforced capabilities for the interface. However, Spin relies on the compiler for enforcement. Spin kernel extensions must be written in Modula-3, which enforces not only the isolation between modules, but also the unforgeability of capabilities required to link with the kernel. This implies a certain level of trust on the source of these kernel extensions. Also, this requires porting of applications from userspace to specialised kernel modules, otherwise cost is still incurred.

[9]Or more fairly two such queues, one for requests, for one responses. They use the same memory for both, however.

CheriOS can link arbitrary programs, written in any language, because it benefits from hardware enforcement for isolation. This not only means there is less work to get a program to run on CheriOS, but also means system maintainers do not need to be as concerned with the source of programs they run.

Opal Opal[22] is a single-address-space system built on top of the Mach microkernel. Like CheriOS, it promotes the sharing of complex, sparse, data structures between programs. Opal relies on the MMU for protection, so the granularity of sharing is page-based, and still relatively coarse-grained. This means that although data can be shared without serialisation, programmers have to pay particular attention to where objects reside in order to not accidentally share data in an unintended way. Like in CheriOS, the OS forces no restrictions on the sharing of either capabilities or pointers between programs, so they are free to construct their own sharing mechanisms. Opal uses 256-bit password capabilities to authorise access to segments. This means that no specialised hardware is required, but also means that read-only side channels can be used to fabricate capabilities. Pointers, however, are not capabilities in Opal. Although authorisation to access a segment is delegated via capability, data references will be transferred separately. This leaves Opal open to the same confused-deputy attacks that other ambient authority systems suffer from.

Userspace networking

Sandstorm and Namestorm Conventionally, application logic sits on one side of the MMU-enforced boundary, while the networking stack and device drivers sit on the other. This is a good split for generality and software-engineering purposes, but not for performance or security. Sandstorm and Namestorm[85] are web and DNS servers that tightly integrate a userspace networking stack. Such integration allows for better exploiting application-layer knowledge, and reduces overheads by merging memory and event models. They demonstrate 2-10x (Sandstorm) and 9x (Namestorm) performance improvements. The downside is a duplication of effort, as a workload that was previously performed by the OS is now performed by the application. Furthermore, isolation of the two workloads is now lost. CheriOS can achieve tighter integration of software (for the sake of performance) without a loss of isolation due to the flexibly of its compartmentalisation.

Marinos et al. report that one of the biggest performance improvements they observe for Namestorm comes from batching system calls across the system-call interface. CheriOS, similarly, tends to batch its IO operations due to the design of its socket library.

Arrakis Arrakis,[103] an OS based on Barrelfish, improves IO performance by divorcing the control plane from the data plane. In userspace, each application hosts its own data-plane IO stack, exploiting better locality, fewer context switches, and application-specific knowledge to deliver better performance. This is done utilising the same virtualisation technology normally reserved for hypervisors with multiple guest OSs. This is therefore only possible when hardware

contains demultiplexing facilities. They show a 2x to 5x decrease in latency, and 9x throughput improvement with a NoSQL workload.

CheriOS sockets, like Arrakis, divorce the control plane from the data plane. Users get direct access to the data plane (although not in the same format as hardware descriptor rings, unlike Arrakis), while leaving the control plane to the OS. CheriOS goes a step further for security, however, also denying most of the OS access to the data-plane.

CheriOS eliminates some domain crossings with its single-address-space shared-memory queues, but unlike these userspace networking solutions, does not go so far as to move networking logic into applications. This is partly because, with much cheaper domain crossings, it may not be unnecessary to do so. On CheriOS, applications can inspect network stack internal state using either read-only capabilities, or fast-leaf calls, both with the same cost as a standard function call. It is therefore unnecessary to import all networking logic into userspace simply to take advantage of application-specific knowledge, because state is shareable in a fine-grained way without loss of security. Of course, such sharing must be built into networking stacks ahead of time, and so does not have the same flexibility as full userspace networking.

6.5 Conclusion

CheriOS offers a secure yet flexible IPC solution due to its use of fine-grained capabilities. This is only possible in a performant fashion because of CHERI hardware. Rather than eschew encapsulation altogether, or use costly call-based mechanisms, CheriOS shares rich data structures between compartments, but imposes arbitrarily complex restrictions on their use by way of user-defined capabilities. This allows a best-of-both-worlds approach, sharing wherever useful, but still enforcing hard boundaries for security and abstraction. When running the same application (NGINX), CheriOS can achieve between 3x and 6x improvement on HTTP transfer rates over CheriBSD, all the while offering a greater degree of security.

Chapter 7

Conclusion

This dissertation has argued that secure systems should be built upon mutual distrust between applications and operating systems. The compartmentalisation model required for such a system is enforceable using capability hardware (such as the novel CHERI architecture), making sparing use of a small software hypervisor (a nanokernel) for purposes such as attestation. A proof-of-concept OS, CheriOS, was developed to demonstrate this idea. CheriOS shows how systems can be designed when both low trust and a high degree of compartmentalisation, without a sacrifice to performance.

7.1 Low trust

CheriOS manages to run with a minimal TCB, on a scale even smaller than a conventional microkernel, resulting in an OS where every other component can be untrusted. The system itself properly exhibits least privilege; system components hold non-equivalent sets of capabilities and no component needs hold all the power. Programs can have their own private memory to run in, with guaranteed confidentially and integrity with respect to other compartments. The CheriOS ABI allows fast transitions to be made between numerous mutually distrusting compartments, and capability- and data-flow between them is restricted to only what programmers explicitly release. CheriOS offers simple yet powerful primitives for program-identity based attestation and capability exchange.

Memory safety, both spatial and temporal, is enforced globally even when the MMU is considered. Programs also enforce control-flow integrity between their compartment and others even if other compartments try to disrupt it. Because of this, individual CheriOS compartments are far safer when running C than programs running on conventional systems.

7.2 Capability IPC

Traditional coarse-grained access-control mechanisms, such as separate address spaces, add increased complexity, limit flexibility, and incur performance costs. Using a purely capability-

based interface in a single address space, such as described in Chapter 6, allows for fine-grained, authenticated access control that can be configured by user applications and enforced by hardware, without trust of the OS. Not only does the design offer flexibility and security, but it can perform well. CheriOS exhibits significant improvements over FreeBSD for HTTP workloads (although this has only been demonstrated on a very small-scale system), achieving a speed up on the order of 3x to 6x. This overshadows the costs of improved security which cost on the order of tens of percents.

7.3 Future work

Some of the techniques used in CheriOS, such as slinky stacks, could be equally well applied to other systems. Applying them to other systems remains future work. One of the largest limitations that CheriOS currently has is its allocation strategy for the heap, which is a result of there existing no good fine-grained hardware-accelerated revocation mechanism. Fragmentation is currently unavoidable for certain applications. Adopting the fine-grained strategies employed on CheriBSD[138] is also future work. Benchmarking CheriOS with DMA is also future work, as it was not currently possible with the experimental hardware available. To be truly low trust, DMA will also need to be constrained by CHERI capabilities. A proposal for handling this is given in appendix D, but prototyping and measuring the performance of CHERI DMA extensions in CheriOS is ongoing work.

The nanokernel has been designed to have minimal state and limit use of difficult to verify mechanisms (such as call stacks, memory allocations and unbounded computations). It is also trusted, so makes an excellent choice for formal specification and verification. This is left as future work.

CheriOS was designed to run on multicore machines, but FPGA benchmarking has only been performed on a single core implementations due to hardware constraints. Future work should explore if it maintains its good performance when many-core CHERI CPUs are available, as it is a goal of CheriOS to show that security need not come as a sacrifice to performance. This work should also include a larger survey of workloads. CheriOS currently only runs on CHERI-MIPS. Porting CheriOS to a contemporary architecture, such as Arm's Morello[92][55], might be an interesting next step.

Software compatibility is a primary limiting factor when it comes to adoption of new systems and CheriOS is not POSIX compliant. An interesting midway step might be running a hypervisor under the CheriOS nanokernel, and then running a conventional UNIX alongside the CheriOS proper. Some work would be required to offer multiple address spaces (which could not share capabilities) in the nanokernel, but should not otherwise require modifying the design substantially. Certain programs, or program compartments, could then be moved to CheriOS to enjoy an improvement in security and performance, while still supporting a large body of software on the conventional OS.

Appendix A

Interleave-Ordered trees

One common way of storing trees is to walk through its nodes in a given order and serialise the contents into an array. This makes the tree both indexable and compact, at the cost of making it harder to merge trees or take a subtree. This only leaves the choice of which order should be chosen. We might choose an order that makes particular operations faster to implement with real hardware. Here we will concern ourselves with binary trees, but most of these concepts extend to a higher branching factor.

The operation we will optimise for is performing operations on multiple nodes in the same row, where the operation to be performed is a function of each nodes neighbourhood. We will

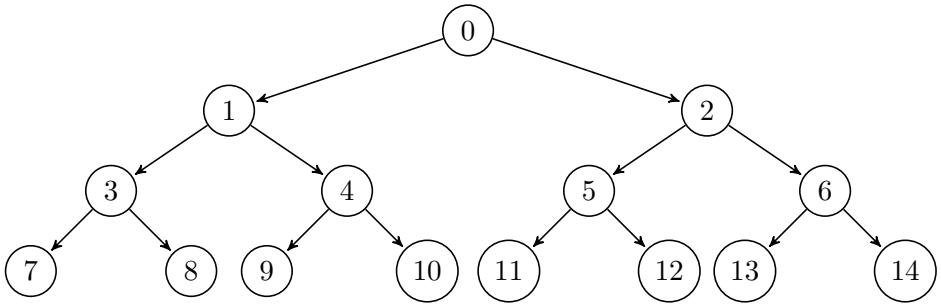


Figure A.1: *Breadth-first order*

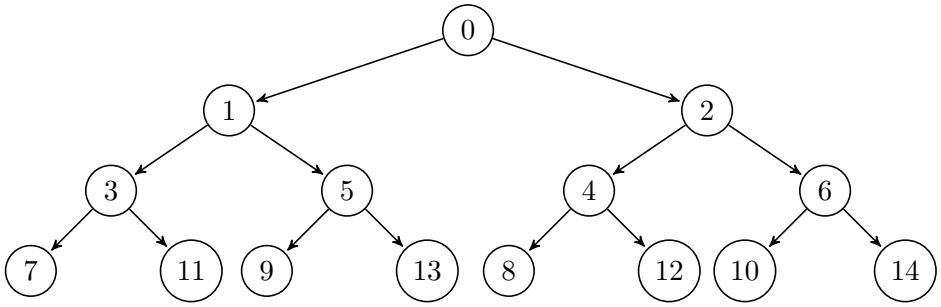


Figure A.2: *Interleave order*

attempt to arrange the tree so that we can use vector operations to manipulate several nodes at once.

Breadth-first, see figure A.1, is a common order used for tree serialisation. If this order is chosen then for a node at index i, its left child is at index $2i + 1$ and its right child at $2i + 2$. Breadth-first has the benefit that nodes in the same row form one contiguous block. The issue with this choice is that even for nodes in the same row, the index difference between it and a child is not the same. Consider the second row. The left child of '1' is at an offset of 2, but the left child of '2' is at on offset of 3.

An alternative is a slight alternation to bread-first which we will call 'interleave order', see figure A.2. Interleave order still maintains the property that nodes on a given row form a contiguous block. However, it also has the property that, for 2 nodes on a given row, the index offset between them and a given child is the same. Again consider the second row. Both nodes have a difference of 2 from their left child, and 4 from their right child. In general, given a node at index i and at depth d (the first row is row 0), its left child is at index $i + 2^d$ and its right child at $i + 2^{d+1}$. Remembering the depth is probably the best tactic when walking the tree, but if only the index is available then $d = \lfloor \log_2(i + 1) \rfloor$.

To see why this order is so beneficial, consider we wanted to set every node at row 2 to the sum of its children. We can load a 4 element vector from locations (7,8,9,10) and another from (11,12,13,14), perform a vector addition, then store the result to (3,4,5,6). This would set element 3 to 7 plus 11, 4 to 8 plus 12, 5 to 9 plus 10, and 6 to 10 plus 14. This is exactly as desired for interleave ordering, and required no permutation of elements in vectors, and did not require scatter or gather, only contiguous loads. Even if a row is larger than a vector, a speed up can be gained by a factor of how many nodes fit a vector. Difference between siblings is also the same, so operations like swapping two children can also be vectorised across rows. In fact, starting with any node in a particular row and then walking any path[1] will give a index delta that is the same regardless of which node in the row we started with.

It may not immediately be obvious why to call this interleave ordering. The name comes from how one would merge two trees stored in this way. To merge two trees stored in arrays in this way one must interleave the elements of the array, and append a new root element. This is obviously somewhat costly, but merging trees stored in arrays is always a costly operation if nodes are to have a deterministic order. To make a tree from the left sub-tree one should take every odd element of an array, and for the right sub-tree every even but element 0.

CheriOS makes use of binary trees with binary elements in one of its compiler passes, and so can pack 64 nodes in a 64-bit register. Interleave ordering is used so that multiple tree operations can be performed using only masks, shifts, and binary operations.

[1] Distinguishing between 'Parent of when left child' and 'Parent of when right child'

Appendix B

CheriOS ABI

This appendix contains the details on how cross compartment calls are implemented in CheriOS.

The guard variable of a DLS is used to make any CFI checks. Currently, only the first 64 bits of the guard slot are in use, the low byte of which is an enum of types. The remaining 56 bits are interpreted depending on the type. The types that are currently in use are:

- **callable_taken**
- **callable_ready**
- **returnable_ready**
- **returnable_used**
- **user_type**

The first four are used for the calling convention, the high 56 bits in them represent a call stack depth. We will write **callable_taken**(X) to mean a guard value with type **callable_taken** with high bits of 'X'. The last type is for programmers to use for their own objects, and the high bits can be interpreted as desired to specify further types of objects. Compiler generated code will often transition objects between the states to do with calling conventions. If this cannot be done, the runtime triggers a trap.

B.1 Caller side

When emitting code at the compilation stage, we cannot yet tell if the function we are calling is in another compartment. Instead, we insert a call using a standard jump-and-link, rather than a CCall, assuming we are making an intra-compartment call. Then, at static link time, we link calls to PLT (procedure linkage table) stubs that implement more complicated calling conventions.

Regardless of trust mode, each PLT stub does the same thing. This is because trust relations are configured dynamically, and stubs are laid out statically. Each stub loads from the captable: the target address for the function, the target data for the function's compartment, and the target mode for that compartment. It then jumps to the target mode. The target mode is a per target-compartment function that the caller uses to select its trust mode. Because this function pointer is stored in the captable, we can dynamically select a trust mode for every process/thread that uses the same library, even if they share the same static code.

The default mode stub is selected depending on compilation flags. Users can also override default behaviour and install different or custom mode stubs for a particular target compartment, if they desire. There are currently four standard mode stubs provided by the runtime, which implement a call with decreasing levels of trust.

The first, called 'same domain', is used when the target is in the same compartment. It is just a CCall.

The second is called 'complete trust'. This is used when a compartment trusts every other compartment. It saves stacks pointers before making a CCall, to be used either on return or on a callback into the compartment.

The third is called 'trusting'. This is used when a compartment trusts the compartment it is calling, but does not trust others. It not only saves the stack registers that 'complete trust' does, but also transitions the thread's DLS guard from **callable_taken**(X) to **callable_ready**(X+1). This is needed as when untrusted callers call into the compartment it will need to be in the **callable_ready** state. It also transitions the DLS state back to **callable_taken**(X) on return.

Finally, the most complex mode is called 'untrusting'. With an untrusting call, the caller does not trust the callee to:

- Save callee-save registers.
- Not read non-explicit argument registers.
- Return to the correct address (with the correct return data), return in order, return only once, and return only when the thread is not already in the compartment.

All of these conventions need enforcing by the caller instead. The first two of these are about protecting confidentiality and integrity of data, the last about control-flow-integrity. The caller saves its state and, along with a return stub that will perform control-flow-integrity checks, forms a continuation that should be CCalled to return. A diagram of how data should look on the stack is given in figure B.1. As it will be sealed with the domain sealer it has a guard variable.

Pseudo-code for the untrusting stub is as follows:

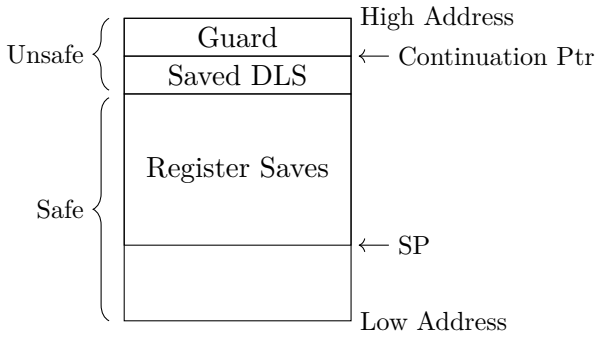


Figure B.1: *Return continuation allocated on the stack. Calls back into the domain to call new functions would allocate new frames after SP. The continuation pointer is sealed when given out.*

```
// Calls the function represented by target_address and target_data without
    trusting the target to obey the ABI

untrusting_stub(code target_address, data target_data) {

  // Allocate space on the stack

  _unsafe guard continuation_guard;
  _unsafe DLS saved_dls;
  _safe register_bank saves;

  // Construct return continuation

  continuation_guard = {.type = returnable_ready, .depth = IDC->guard.depth};
  saved_dls = IDC;

  RETURN_DATA = seal(&guard, IDC->DS);
  RETURN_CODE = seal(&return_point, IDC->DS);

  // Save registers

  save_callee_saves(&saves);
  IDC->SP = SP;
  IDC->USP = USP;

  // Set threads DLS to ready for callbacks

  IDC->guard = {.type = callable_ready, .depth = IDC->guard.depth+1}

  // Zero most registers and then CCall

  zero_registers();
  CCALL(target_address, target_data);

  // Callee should return here (IDC should point to the stack allocations)
  return_point:

  // Check this is a return continuation and has not been used
```

```
  int X = IDC->continuation_guard.depth;
  int success = CAS(&IDC->continuation_guard, {.type = returnable_ready, .depth =
    X}, {.type = returnable_taken, .depth = 0});
  if(!success) trap();

  // Also check that the return order is correct, make sure the thread is not
    already in the compartment, and block further entrance

  success = CAS(&IDC->saved_dls->guard,  {.type = callable_ready, .depth = X+1},
    {.type = callable_taken, .depth = X});
  if(!success) trap();

  // At this point the return is guaranteed to be legal. Now restore callee saves.

  restore_callee_saves(IDC->saves);

  // Restore IDC to threads DLS

  IDC = IDC->saved_dls;

  // restore stack and global pointers

  SP = IDC->SP;
  USP = IDC->USP;
  GP = IDC->GP;

  return;
}
```

Some of the stack allocations made are unsafe, as a callee might try to return after the lifetime of the stack frame, see section 4.1. Once saved, registers are cleared, ensuring no extra capabilities are passed other than intended.

After the call has been made, there are only two ways to return to the compartment as the same thread: either CCall-ing the return continuation we just constructed, or by calling into another function in the compartment. See section B.2 for how the second case is handled.

When the return continuation is CCalled, we will jump back to the second half of the untrusting mode stub, and IDC should point to the stack allocations. Two atomic checks are made to ensure correct compartment-boundary CFI. The depth field on the guard is used to make sure returns are being made in the correct order. Once both atomic checks have been made, all callee saves are restored.

It should be noted that this mechanism is somewhat heavyweight compared to a standard function call. However, it provides the same level of isolation one would expect from an IPC mechanism, or from a syscall into the kernel. If we compare the cost of this untrusting call to even fast path IPC calls on other OSs, it seems far more appealing. See section 5.4 for a discussion on the costs of transitions.

B.2 Callee side

With an untrusted call, the callee does not trust the caller to:

- Jump to a proper target address (with the correct target data), and jump only while the thread of control is not already in the compartment.
- Not read non-explicit return registers on return from the call.

As with the caller side, the callee will have to enforce these properties itself. It will not want to make the effort for intra-compartment calls, however, and so in order to provide different functionality every function is given three different entry points:

- A standard entry for intra-compartment calls.
- A trusted entry for when all inter-compartment calls are trusted.
- An untrusted entry for untrusted inter-compartment calls.

The standard entry does no extra work. It assumes that the stack, unsafe stack, and globals pointers will all be set up sensibly. Unsealed pointers to this entry are put in the captable for intra-compartment calls. The trusted entry loads the stack pointer, unsafe stack pointer and globals pointer from the DLS and then falls through to the standard entry. Pointers to trusted entries are also provided unsealed, but to other compartments. The untrusted entry, however, needs to do more. Firstly, the address of the untrusted entry, as well as the data argument for the compartment, are both sealed with the domain sealer before being published. This ensures that other compartments can only transition to a proper set of entry points. On a call, we still need to check that a correct data argument was used, and that a thread of control is not already in the compartment.

In order to not bloat code, the untrusted entry performs a jump-and-link to the DL (domain link) loaded from the DLS, which will jump back when it has done this extra work. The standard untrusted domain link will first check there is no thread already acting as the current one in the compartment, and the target data installed in IDC is of the correct type. It will do this by tying to transition the guard from **callable_ready**(X) to **callable_taken**(X). Next, it stores and modifies the return address to insert a return stub to be called on the return edge of the originally called function. It then calls the trusted entry of the intended function.

On the return path, the untrusted stub transitions back from **callable_taken**(X) to **callable_ready** (X), and restores the correct return address and return data. Finally, it clears every register apart from the return address, return data, callee saves, and return registers. It then CCalls back. It is important to note that, even though we do not trust our caller, we still save callee saves. A trusting

caller would still expect this. A correctly functioning program cannot leak data through its callee saves on return as their values should be unchanged, and therefore convey no information.

One important trick is that the standard function return always uses CCall. This means the callee does not need to know if a trusting or untrusting call was made on the callers side; it does the same thing regardless. This is also true on the callers side, CCall is always used in PLT stubs and so different code paths are not required depending.

Those familiar with C semantics may at this point note that having multiple pointers for the same function is at odds with the need for pointers to the same function to compare equal, even if their address was taken in different libraries. For the most part, target address/data pairs are hidden by PLT stubs. Sometimes, a programmer may wish to use them explicitly, and in the current implementation we force programmers to request a cross-domain function pointer if they want to use one manually, which will give a code/data pair not comparable to a single pointer. These pairs can be compared using a provided function which knows about extra entry points, and returns whether or not the pair represent the same function, regardless of their trustedness. This pair can be called using an invocation function that knows how to short circuit the call if it came from the same compartment, or otherwise uses a specified trust mode. Further compiler support to make cross-domain function pointers equivalent to function pointers is future work.

Appendix C

CHERI-aware escape analysis implementation

Slinky stacks (discussed in section 4.1) are a new stack allocation structure that support temporal safety via selective non-reuse. The rate at which slinky stacks consume memory depends on the cumulative size of the unsafe allocations needed. If a function makes no unsafe allocations, slinky stacks use as much memory as standard stacks. The CheriOS C compiler offers 'safe' and 'unsafe' keywords. Programmers can manually declare a variable's temporal safety as required, and also specify with a compiler flag how untagged variables should be allocated: Either all safe for programs that have had proper vetting, or all unsafe where a programmer desires higher assurances.

Although useful, this alone is insufficient. The cost of annotating old code bases is unlikely to be paid, and the chances that a programmer makes a mistake when annotating are too high. Manual tagging should only be used sparingly, where optimisation is strictly necessary and a programmer is willing to make special effort to ensure correctness. Instead of manual tagging, we employ static analysis to tag allocations as safe and unsafe without programmer assistance. CheriOS's only use of manual tagging is to declare allocations passed to nanokernel functions as safe. This is done as several nanokernel functions return by reference, and there is no point in treating a part of the TCB as an adversarial compartment. Manual tagging completely overrides the automated pass, even if the two disagree. In the previous example, analysis will correctly deduce that a reference to a stack allocation is being passed across a protection-domain boundary. Manual tagging intentionally overrides this. CheriOS does not use the unsafe keyword, it is only useful when default-safe is used and the automated pass is off.

We perform our analysis during compilation at the LLVM-IR level. This was chosen over the C level as we also want to protect against undefined behaviour, which accessing variables out of their lifetime is. We want it to be the case that any C program that compiles, as long as it does not use inline assembly (which is very hard to reason about), will be safe, regardless of how much undefined behaviour was employed. The analysis runs purely in LLVM during compilation; it need not run on CheriOS and so is not a part of the run-time TCB.

LLVM has a capture-tracking analysis pass (a pointer value is captured if any part of the pointer outlives the call), which it uses to promote stack allocations to registers. This promotion allows for further optimisations, and the majority of stack allocations are already promoted in this way. However, we can do better in a few ways than LLVM can. Firstly, we are not interested in whether the address of a stack allocation is captured (which LLVM's capture tracking is concerned with). Instead, we are interested in whether the *capability* to access that address escapes either the current compartment, or the lifetime of the function. Secondly, LLVM IR has enough information to statically verify that some values cannot be capabilities. If something is not a capability then it cannot be used to access an allocation, which helps limit what IR values need be considered. Thirdly, CHERI puts strong limitations on how capabilities can be derived, and as pointers are implemented using capabilities, this in turn limits how pointers may be derived. Also, LLVM makes no attempt to track dataflow through loads and stores, whereas the CheriOS analysis pass does.

The analysis we perform in LLVM is a type of alias analysis. However, as we are only interested in aliasing up to temporal safety, we simplify the algorithm by having only two alias classes: 'safe' and 'unsafe'. These classes correspond to those pointers which point to objects safe to allocate on reused stack memory, and those not. The algorithm could be extended to include further classes if a more complete alias analysis were ever desired, for example, for the purpose of other optimisations.

CheriOS's alias analysis works on static LLVM IR, and is not flow or field sensitive. It is also mostly context insensitive, although it does do some static inlining at call sites. Being flow insensitive is necessary, as an adversary with one of the capabilities we are analysing may be making concurrent modifications. We make the following assumptions:

1. At the point of allocation, capabilities to a function's stack allocations do not alias with any others.

2. Capabilities have a source of only a single allocation.

3. A capability aliases with no more than the capability it is derived from.

4. A value that is not a capability cannot alias with anything.

The first two assumptions rely on having memory safety, which our system guarantees. It is not circular to require that we can assume temporal safety in order to provide temporal safety. We only assume temporal safety of *other* functions in our compartment. As long as programmers do not violate temporal safety with bad manual tagging of local variables, and the automatic pass is used on every function in the compartment, then we arrive in a consistent state. One compartment cannot break stack temporal safety for another as they all use isolated and protected stacks, and all inputs to compartments are considered unsafe. However, if a programmer has already broken temporal safety with bad manual tagging then this can lead to further unsafety. The latter three rely on CHERI properties. CHERI instructions are all single provenance (capability

	X := Y	X := *Y	*X := Y
X is ... Y	Derived from	Loaded from	Stored by
Y is ... X	Used to derive	Loaded by	Stored to

Table C.1: *Capability relationships*

instructions only derive a capability from one input[1]) and monotonic (capability instructions can only produce a capability that is a subset of its input).

There are three kinds of binary relationships between two capabilities that we consider, when combined with which of the two we are considering gives 6 relationships. See table C.1. As we are not field sensitive, all pointer manipulation counts as assignment. Manipulating a capability cannot make it refer to a different alias class, as this would violate either memory safety or monotonicity. As we are single provenance, expressions such as X = f(Y, Z), where f is a function or intrinsic that performs no loads, stores, or global accesses can be broken down into X = Y and X = Z. X could be either of the two, but not both. As the analysis is flow insensitive, there is no ordering between the two assignments. In this way, all intrinsics that deal with manipulating capabilities, such as modifying bounds or permissions, or intrinsics such as phis and selects, can be broken down and just considered as assignments.

It is important to think about X = Y as two relations when it may first appear there should be some symmetry. Consider the following example:

```
B = φ(A, C);
...
D = C;
```

Consider what A might alias with. It could alias with B as we have B = A from the phi function. B can also alias with C from the phi function, and C in turn can also alias with D from the second line. Together this means that A can alias with B, B can alias with C, and C can alias with D. However, this does not imply that A can alias with D. Because of single provenance (phi is always single provenance, regardless of CHERI, but we could have the same example with a different function), B can only be A *or* C. In the case it is A, it is not C, and therefore not D. It is also not as simple as only following assignments in one direction. B *can* alias with D, and to show this we follow the first assignment from left to right, and the second right to left. The CHERI alias analysis will capture this asymmetry exactly. Loads and stores have a very similar asymmetry for aliasing as derivation. Any load from a location may alias with any other load (the value may not have changed). Furthermore, any store may alias with any load. However, this does not imply that stores to the same location become aliased with each other.

In LLVM, every variable or result of an instruction is an 'LLVM value'. Some of the LLVM values are capability-typed. If a LLVM value is not capability-typed the instruction or register

[1]The truth of this statement depends on if a sealed capability is considered to be derived from the capability that sealed it. If we consider this to be true, then sealing derives a capability from two, but still only a maximum of one *memory* capability, as is required to only have provenance of a single allocation.

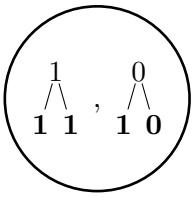


Figure C.1: *An example safety label for a variable. Boldface indicates the binary tree continues with nodes of the same value forever. This label means "Any variable derived from this one is at least as unsafe as the first tree, and any variable this one derives from is at least as unsafe as the second". The first tree means "This variable is unsafe, and anything loaded or stored to it should also be unsafe". The second tree means "This variable and anything stored to it is safe, but anything loaded from it should be unsafe".*

it will eventually be lowered to will be unable to generate or be a capability. This is because instructions are explicit in whether they take or produce a capability or an integer. C's `uintptr_t` and friends are considered to be capabilities by LLVM. Casts from capabilities to integers result in tag stripping. For every capability-typed LLVM value we create a corresponding node in a graph and label it with alias information. Once the pass has finished we will use the alias label attached to each node to make judgements about the corresponding LLVM value. We can ignore non-capability LLVM values due to assumption 4, they carry no privilege. For example, bits extracted via comparisons or side effects of stores can be ignored; They may convey information but not privilege. In the end, we are only interested if the value is unsafe (i.e. escapes), and so only consider two alias classes: the safe class and the unsafe class. Although eventually we will assign each alloca to one of the two classes, 'safe' or 'unsafe', the label we store for each node is not just the class it belongs to. Instead, we store more information to also track what classes it may point to, and to capture the asymmetry of the 'derives from' / 'used to derive' relationship.

We initialise the nodes label depending on what kind of LLVM value the node represents, e.g. is it an alloca or is it an argument, and then apply a fixed point algorithm that applies constraints on nodes across edges in the graph. Each edge in graph represents a use of the two values in conjunction. See table C.1 again. To represent the asymmetry around derivation, every node's label will be pair (X, Y). Notionally, X is used to propagate the 'Used to derive' relationship, and Y for the 'Is Derived From' relationship. X and Y are themselves both binary trees.

These binary trees represent the classes that may be pointed to. Each node in a tree is either a '1', which indicates being unsafe, or a '0', which indicates being safe. If the analysis were extended to arbitrary classes then the nodes would instead be a set of classes. The root of the tree is what class the pointer itself belongs to. The left sub-tree is what class a value loaded from a pointer may alias with, and the right sub-tree is what class a value stored to a pointer may alias with. The tree is conceptually infinite, a capability can be dereferenced any number of times. If a '1' occurs in the tree, every child of that node must also be a '1'. That is, if a pointer is unsafe, everything it points to is unsafe and so on. Figure C.1 shows an example of one of these labels.

Initially, we label all nodes that corresponding to globals, input arguments, returned values, values returned from a called function, and values passed to a function as 'maximally unsafe'. The maximally unsafe label is a pair of trees where every node is 1. If we have static meta-data about a function in the same compartment we might be able to relax this slightly. We initialise every other node with a maximally safe label. This is a pair of trees where every node is 0. For this to be a correct initial state the compiler should be zeroing stack allocations if they are potentially read before being written. Many compilers already have this feature, and reducing its costs on modern hardware have been explored before[140], as such, CheriOS does not specially implement it. If this guarantee cannot be made, then the constraints stay the same but the graph needs to be initialised differently. Rather than starting in the maximally safe state, allocas should start with the roots of their trees being 0, but every other node a 1, which signifies anything loaded from or stored to the allocation may be unsafe. It may be be obvious that values loaded from un-initialised memory may be unsafe, but stores are slightly less obvious. If we store to a stack location, a future function may make an uninitialised load and then leak it, and if other functions in our compartment may do this we need to consider stores to stack variables unsafe.

The next step is to propagate constraints across edges until no further changes occur. When a constraint is propagated a nodes new label is the union of the label it already had and the new constraint. The union of two trees is node-wise binary OR of the two. The union of two labels is a pairwise union of both trees. A graph showing the six types of propagated constraint is given in figure C.2. Note a few things: first, the asymmetry between 'Used to Derive' and 'Derived From'. 'Derived From' sets both trees, and 'Used to Derive' sets only the first. This will allow propagation through 'Derive From' followed by 'Used to Derive', but not the opposite. Secondly, note the difference between 'Loaded From' and 'Stored By'. 'Stored By' does not set the right-hand of the tree, as stores do not impose constraints on other stores. Loads set both the left-hand side and right-hand side of the tree as loads effect other loads and stores.

Two other features are quite subtle: the empty second tree in 'Loaded By' and the use of the weaker 'B' rather than 'A' in 'Loaded From'. This is done to stop feedback between the two. LLVM IR is SSA (Static Single Assignment). This means that, statically, a value can only be the result of a load or a derivation, not both. Thus, from the set of the edges that start at a given node there cannot be both a 'Derived From' and 'Loaded From' edge. Consider a node with a 'Loaded From' edge leaving it. The existence of the edge means there will be no 'Used To Derive' edges coming in to the node. As the only edges that create any asymmetry between the pair of trees are 'Loaded By' (which we don't want feedback from) and 'Used to Derive' (which won't exist) it is safe to use the second tree. Similarly, consider a node that is the target of a 'Loaded By' edge. The only edges that would use the second tree of the pair are 'Derived From' and 'Loaded From'. Again we don't want feedback between the two load related edges, and the target of a 'Loaded By' cannot be 'Derived From' anything because of SSA. Feedback between the two loaded related edges would otherwise remove any differences between the pair of trees. This is due to the trees themselves not storing pairs in their nodes, only a single value.

Finally, once a fixed point is reached, the LLVM IR allocas that have a '1' in the root of the first tree in their node's label are tagged 'unsafe' allocations, or 'safe' allocations otherwise.

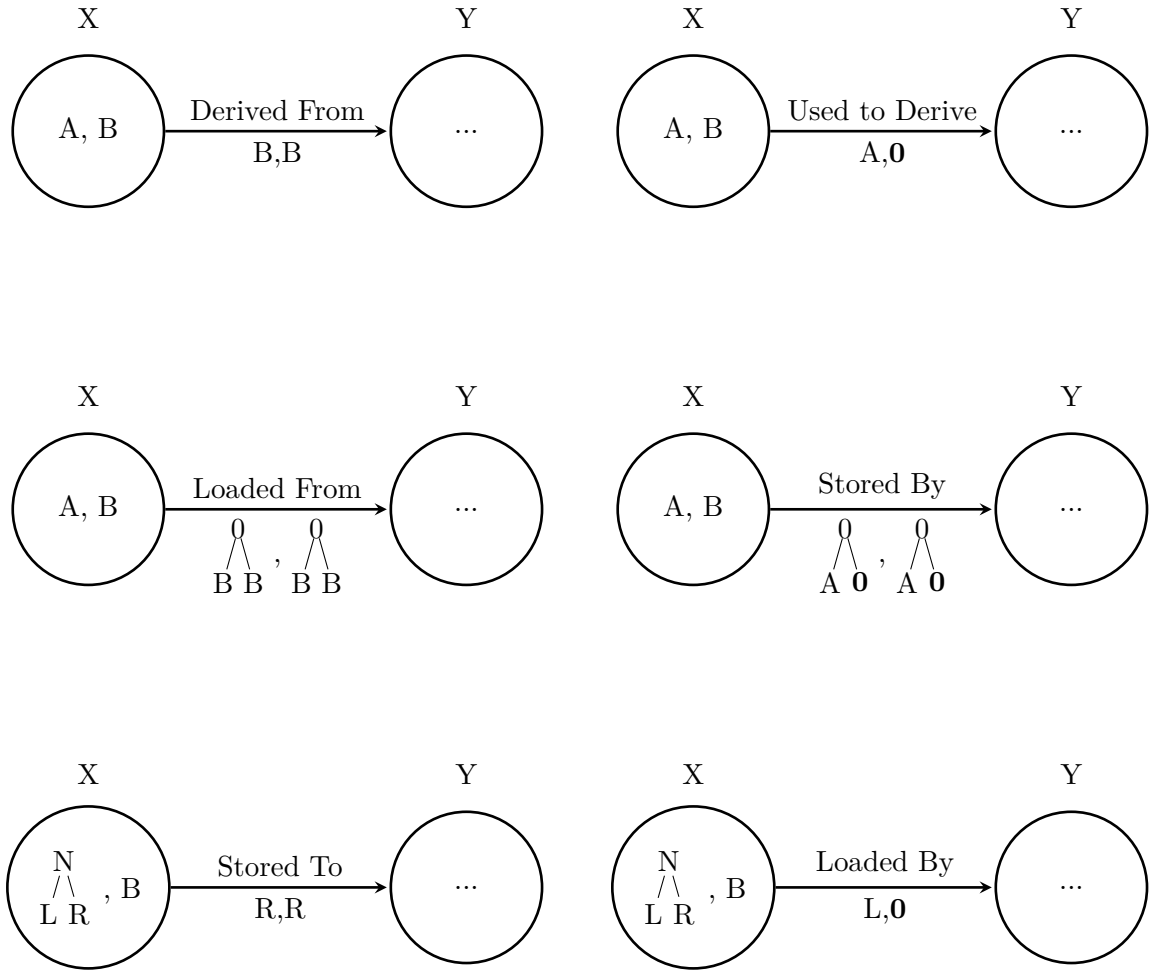


Figure C.2: *Constraint propagation graph for each of the six relationships. Nodes represent program source variables. Labels represent alias information. Edges are generated for usages of variables in conjunction (X is ... Y). The top of each edge is labelled with the relationship and the bottom with the propagated constraint. If the propagated constraint were Q, and the node it arrives at had label P, the new label would be Q⋃P. Boldface* ***0*** *is an infinite tree of all 0s.*

The first tree is used (rather than the union of the two) as it is always a superset of the second due to how constraints are propagated.

In the current implementation the trees are limited to a depth of 6, giving 63 nodes per tree. This number of bits can be packed into a 64 bit integer, which makes node-wise binary or a trivial operation, the 64 bit integers are just or-ed. A depth of 6 means we can track up to 5 indirections deep, which is sufficient for most purposes. In order to keep the analysis sound when (or indeed if) leaves are lost off the bottom of the tree (by appending a new root) we make the parent node a bitwise or of its children. Similarly, when new leaves are created (by taking a LHS or RHS of a tree) we set them equal to the parent. If a graph had 5 indirect stores, followed by 5 indirect loads, asymmetry between load and store sides of the tree would be lost. In C, this would correspond to an expression such as *****(x). In practice, no information is lost with this depth. 5 indirections is a large number to make statically. An interesting order of tree flattening

was chosen to make setting all nodes in a row to a function of their children vectorisable. See appendix A.

A few final optimizations were made. If a function is called that is in the same compartment, rather than consider its return value and every argument we give it to be unsafe, we statically inline it. For every function we analyse, we inline static call sites a few times and a static call tree is created. The number of nodes in the tree are limited, and can be specified on the command line when invoking the compiler. 64 was used as a default. We apply the same 'Derived From' and 'Used to Derive' constraints between arguments at call sites and input arguments of the inlined functions, and also between return expressions in inlined functions and the value returned at the call site. Only globals, the inputs and returns to the root of the call tree, as well as arguments to functions we did not inline are considered unsafe. The alias semantics of in-builts like `memcpy` and `memset` are also provided as they occur so commonly. For example, X = `memcpy`(Y, Z, n) is X = Y, T = *Z and *Y = T, for fresh T. Cross function analysis leads to far better results, see section 4.4. This in-lining is more powerful than trying to compute a type signature for a function as it is context sensitive. Calling a function with two local variables may result in different behaviour than calling it with one global and one local.

Appendix D

CLUTs

CheriOS proposes that an IOMMU be in place to facilitate address translation and also a fast method of revocation in the same way the MMU was used for the CPU, but not as the fundamental access control mechanism. Both the IOMMU and MMU should be filled using the exact same single address space, so that capabilities have the same interpretation to hardware as they do with software. In order to provide the same fine-grained protection as CHERI capabilities, we add a new hardware adapter, called a CLUT (Capablity LookUp Table), that sits between the device and the IOMMU. The device is agnostic to the presence of the CLUT, and so needs no modification, but the device driver is aware of and can directly manage the CLUT. The driver is still untrusted. The CLUT enforces that CHERI capabilities are used for accesses, but then translates requests to plain virtual addresses to be handled by the IOMMU.

A diagram for an access made by CLUT is given in figure D.1. It is, in essence, a per device segment table where entries are capabilities. They are very similar to the tables used in the Burroughs B5000 (for holding segment descriptors) and Cambridge CAP. They are managed by drivers, and translate addresses generated by devices on the fly during DMA. A CLUT is

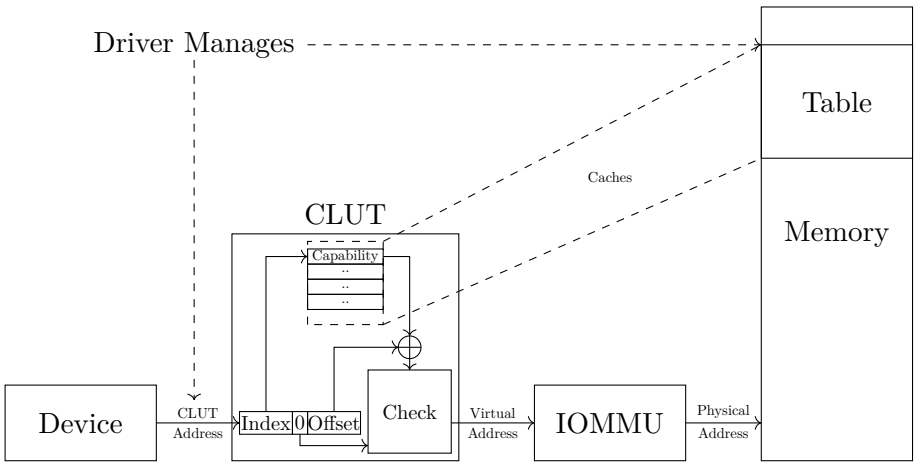


Figure D.1: *A CLUT (Capability Look Up Table) in operation. The driver manages CLUT addresses.*

formed of a small cache (flushable by the host CPU for revocation and coherence purposes) and some memory-mapped registers. The memory-mapped registers are used to control the CLUT. One of them is a capability to a table of capabilities in main memory. This table of capabilities is managed by the device driver, and is used to grant the device virtual memory capabilities. Rather than writing a physical or virtual address to the descriptor rings of a device, a device driver instead writes a CLUT address (which is effectively just a segment index and segment offset appended). It writes any other fields as normal. A *CLUT address* is formed of an index, a guard bit which should always be 0, and an offset. The CLUT translates a CLUT address by indexing the table of capabilities with the index, and then adding to that capability the offset. The resulting capability can then be bounds and permission checked by the CLUT with the same semantics as CPU reads and writes. If the check succeeds, the CLUT forwards the IO operation to the IOMMU but with just the virtual address.

Devices will read the descriptor and then perform a series of IO operations with they think of as a physical address. This results in offsets being added to the CLUT address. Adding to a CLUT address will result in adding to its offset field, and so will give the intended address. The guard bit is there to catch accidental overflows of the offset field into the index. The guard bit should always be 0, and if a guard bit of 1 is given the CLUT will behave as if an out of bounds access was attempted. Obviously, a very large offset would not trigger this check, but non-buggy drivers should not attempt to write descriptors that would result in overflow. The guard bit is not required to ensure that the device does not use a capability it was not granted, the device is allowed to make use of any capabilities the CLUT has been granted.

On a privilege check violation, the CLUT will the stop the access, and all further accesses, and then raise an interrupt. This interrupt can be handled by the system, potentially terminating the misbehaving driver, then resetting the device and CLUT.

Granting a device driver the capability to write descriptor rings and a CLUT's capability table gives it no power to perform memory accesses it could not already perform using CPU operations and those capabilities, albeit more slowly. Furthermore, even misbehaving devices can only access memory they have been granted via CHERI capabilities. We need only trust that the CLUT performs the capability checks correctly. The same CHERI capability used in a user program can be used to perform DMA, directly into or out of the space of the user. This would work particularly well in conjunction with the CheriOS socket layer in achieving zero copy IO between all devices and applications by default. As capabilities used by the CLUT stay resident in main memory they will be caught by the nanokernel's revocation pass.

Appendix E

In-place deduplication and compaction of programs

Applications can automatically take advantage of CheriOS's deduplication services (described in section 5.3) by calling a standard library during initialisation. It is important that this occur from inside the program, as the ability to make arbitrary modifications from an external linker would compromise CheriOS's strong compartmentalisation. The process by which this is done is fragile, as the program is performing self modification while running. There are two steps to deduplication.

The first is to rewrite pointers. A program needs to identify every pointer that exists in the program's global memory, work out what object that pointer corresponds to, and re-point it to deduplicated data if it pointed to constant data and a duplicate version exists. On a traditional system, this is practically impossible as it is hard to tell the difference between pointers and integers, and object bounds are unknown at run time. However, with CHERI, every pointer contains bounds information. At start up, a program will contain no capabilities in any of its segments. During initialisation, a capability relocations table is processed to subset capabilities to individual objects out of capabilities to entire segments. This step is described in section 4.1. Because the program has not started yet, these relocations give a handy list of every capability a program contains in any of its segments. These relocations even describe which segments objects point to, so we can tell which are valid targets for deduplication. The deduplication step re-parses the capability relocation table, and for every capability it finds that points to constant data, tries to deduplicate that data and replace the capability. After this step is complete, future uses of some capabilities, such as those in the captable, will refer to deduplicated versions.

The next step is to compact read-only segments. This step is the most involved, as it runs the risk of interfering with the program as it compacts itself. Once again, the capability relocation table is used as a handy way to find all capabilities and thus all data that can be reached. This time, however, we sort it. The sort order is lexicographic, first on whether the object still points to non-deduplicated data, then on the the base address of the object, then on size. Once sorted, the second half of the list containing the locations of deduplicated objects is discarded, leaving a list of objects that still point to the read-only segment, in order of their base address. Now the

compaction itself is performed. We loop through the sorted list, copying the objects they point to, in order, to the read-only segment. We then fix up the capability to point to the copy, still with correct bounds. It is not possible to overwrite an object that has not yet been copied due to the sorting. Care has to be taken as some objects might point to a subset of another, and copying it twice would be an error. Because objects were sorted by base address then size, we will copy the superset first, and can point subsequent capabilities into the already copied object if their base address is contained within the previously copied object. The result of this step is that the still in use portion of read-only segments are now compact. At present, CheriOS makes no attempt to re-use the space, compaction serves only to get better cache and TLB performance, although memory could also feasibly be unmapped.

There were a few subtleties that needed to be handled with the compaction step. The first was if the compaction function tried to move itself. This is handled by ensuring that the compaction function is always deduplicated in the first step. It is therefore the deduplicated version that gets called, and so it won't be moved. The second problem is that the function that called the compaction function may have been moved, and so the compaction function will return to the wrong address. This is handled by first ensuring that the calling function is the first function on the stack, and then providing the return address as an explicit argument to the compaction function. If the compaction function moves the function that called it, it can modify the return address appropriately. Finally, some read-only data is used by the compaction that might be moved. This means no sub-routines can be used during the copying phase as any routines used may be over-written while in use. This is solved by in-lining all functionality used during the copy step (which was in fact only memcpy). The other piece of read-only data that gets compacted is the relocation table itself! This is needed as only references to the table are actually sorted. If during the copy step the table is moved, the compaction function changes what reference it is using to loop through the list.

Appendix F

Programs running on CheriOS

Here, just for fun, I include some screenshots of applications where the server was hosted on CheriOS.

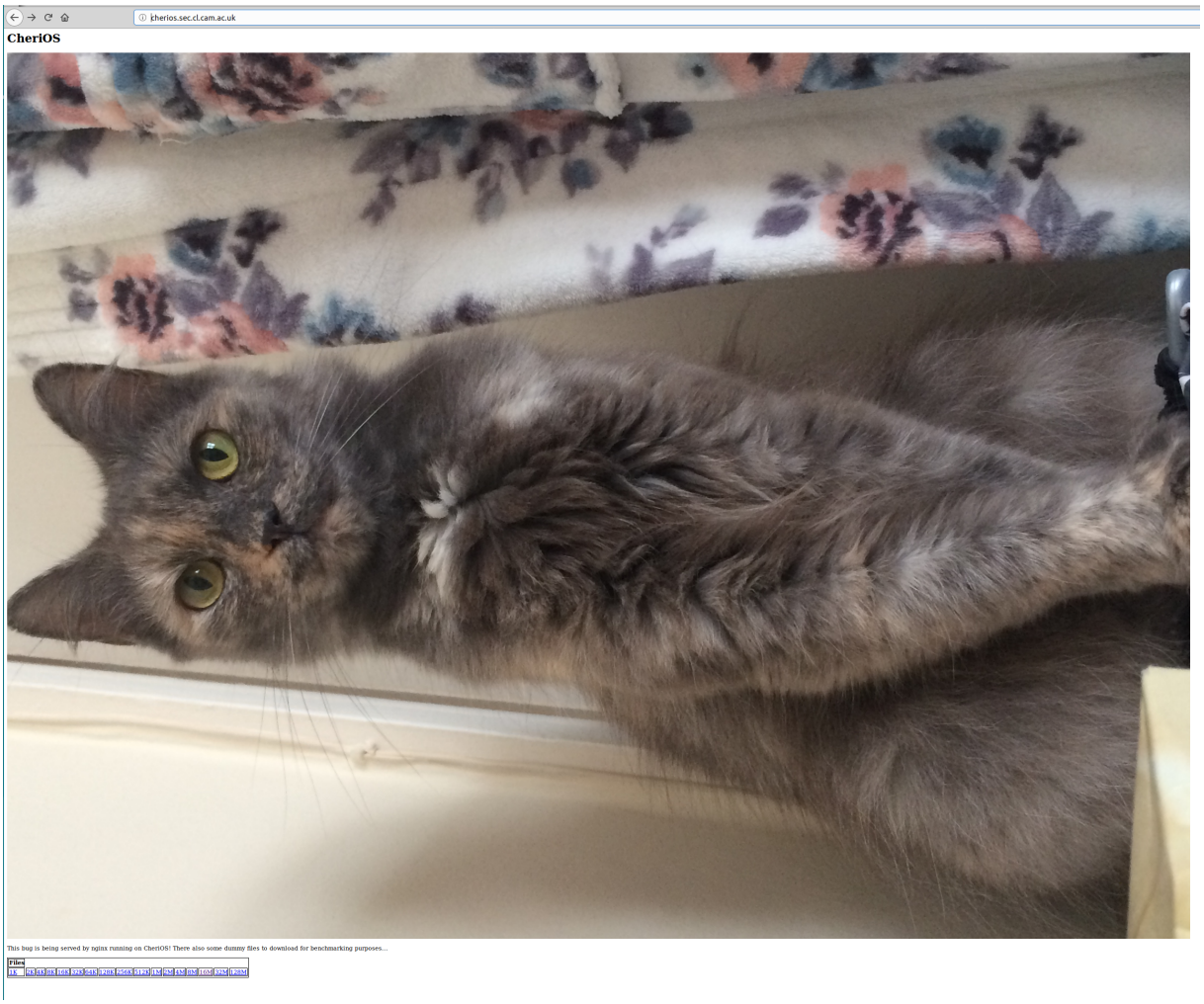


Figure F.1: *Some claim the primary function of the Internet is to serve pictures of cats. In order to demonstrate proper usefulness, here is a screenshot of CheriOS serving a picture of the venerable Bug.*

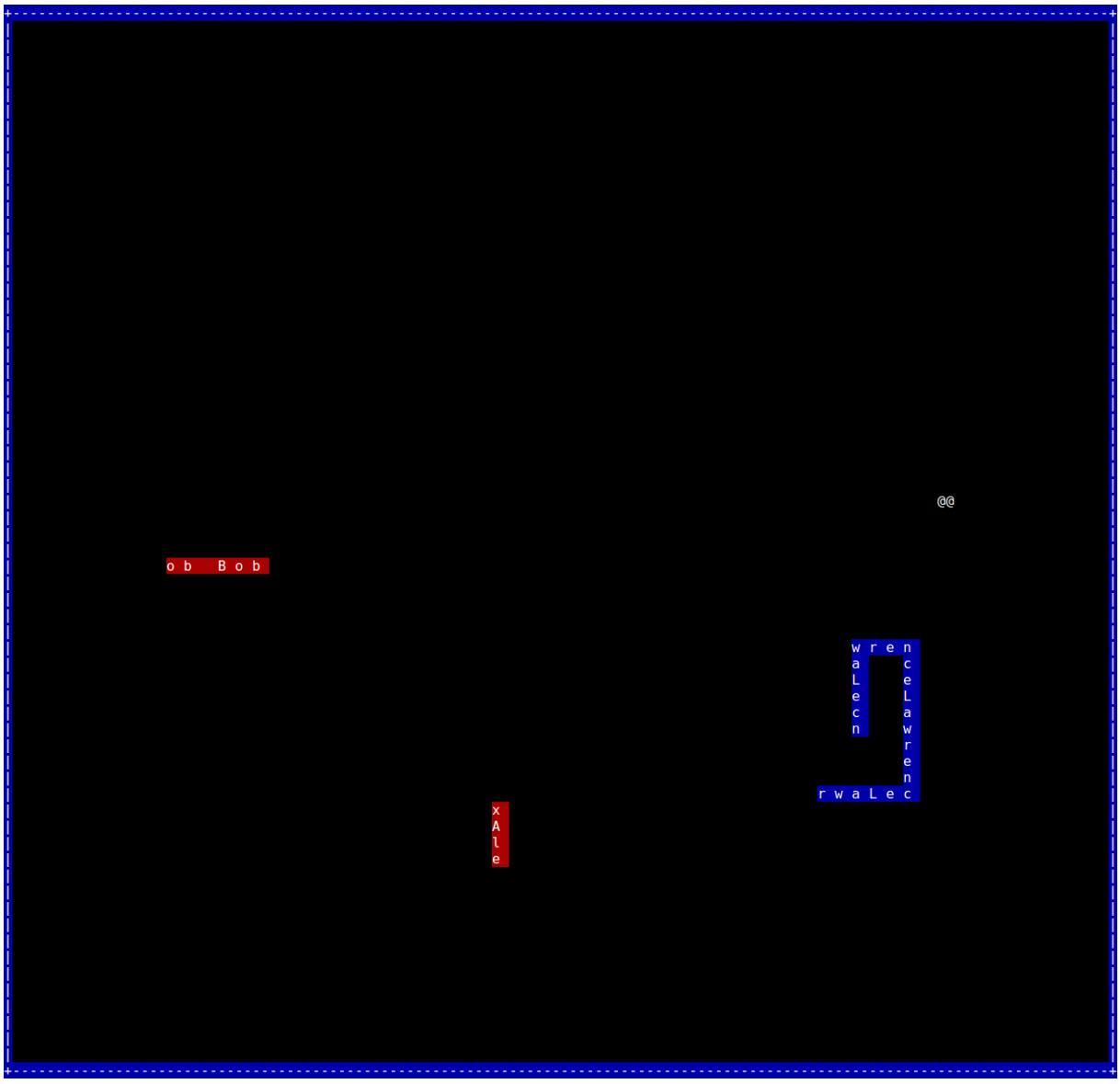


Figure F.2: *A simple multiplayer snake variant hosted on CheriOS and playable through the terminal. One M short of an MMO.*

References

[1] Accetta, Mike et al. 'Mach: A new kernel foundation for UNIX development'. In: (1986).

[2] Aichinger, Barbara. 'DDR memory errors caused by Row Hammer'. In: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2015, pp. 1–5.

[3] Akritidis, Periklis. 'Cling: A Memory Allocator to Mitigate Dangling Pointers.' In: *USENIX Security Symposium*. 2010, pp. 177–192.

[4] Akritidis, Periklis et al. 'Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.' In: *USENIX Security Symposium*. 2009, pp. 51–66.

[5] Almossawi, Ali, Lim, Kelvin and Sinha, Tanmay. 'Analysis tool evaluation: Coverity prevent'. In: *Pittsburgh, PA: Carnegie Mellon University* (2006), pp. 7–11.

[6] AMD. 'AMD SEV-SNP: Strengthening VM Isolationwith Integrity Protection and More'. In: *White paper* (2020).

[7] Anderson, Ross et al. 'Measuring the cost of cybercrime'. In: *The economics of information security and privacy*. Springer, 2013, pp. 265–300.

[8] Anderson, Thomas E et al. 'Scheduler activations: Effective kernel support for the user-level management of parallelism'. In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 53–79.

[9] Andronick, June et al. 'Large-scale formal verification in practice: A process perspective'. In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 1002–1011.

[10] Antonakakis, Manos et al. 'Understanding the Mirai Botnet'. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[11] Arcangeli, Andrea, Eidus, Izik and Wright, Chris. 'Increasing memory density by using KSM'. In: *Proceedings of the linux symposium*. Citeseer. 2009, pp. 19–28.

[12] Barham, Paul et al. 'Xen and the art of virtualization'. In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 164–177.

[13] Baumann, Andrew et al. 'The multikernel: a new OS architecture for scalable multicore systems'. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 29–44.

[14] Bawden, Alan et al. *LISP Machine Progress Report*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1977.

[15] BBC. 'Hospital Hack 'Exploited Heartbleed''. en-GB. In: *BBC News* (Aug. 2014).

[16] Bershad, Brian N et al. 'Extensibility safety and performance in the SPIN operating system'. In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 267–283.

[17] Bomberger, Alan C et al. 'The KeyKOS nanokernel architecture'. In: *In Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*. 1992.

[18] Brasser, Ferdinand et al. 'Software grand exposure:{SGX} cache attacks are practical'. In: *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*. 2017.

[19] Brooks Jr, Frederick P. 'The mythical man-month (anniversary ed.)' In: (1995).

[20] Carter, Nicholas P, Keckler, Stephen W and Dally, William J. 'Hardware support for fast capability-based addressing'. In: *ACM SIGPLAN Notices*. Vol. 29. 11. ACM. 1994, pp. 319–327.

[21] Cass, Stephen. 'The 2015 top ten programming languages'. In: *IEEE Spectrum, July* 20 (2015).

[22] Chase, Jeffrey S et al. 'Sharing and protection in a single-address-space operating system'. In: *ACM Transactions on Computer Systems (TOCS)* 12.4 (1994), pp. 271–307.

[23] Chen, Guoxing et al. 'Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution'. In: *arXiv preprint arXiv:1802.09085* (2018).

[24] Chen, Yue et al. 'Downgrade attack on trustzone'. In: *arXiv preprint arXiv:1707.05082* (2017).

[25] Chou, Andy et al. 'An empirical study of operating systems errors'. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 2001, pp. 73–88.

[26] Cimpanu, Catalin. *Microsoft: 70 percent of all security bugs are memory safety issues*. 2019. URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (visited on 10/05/2019).

[27] Cojocar, Lucian et al. 'Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks'. In: *S&P'19* (2019).

[28] Collberg, Christian S et al. 'SLINKY: Static Linking Reloaded.' In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 309–322.

[29] Colwell, Robert P, Gehringer, Edward F and Jensen, E Douglas. 'Performance effects of architectural complexity in the intel 432'. In: *ACM Transactions on Computer Systems (TOCS)* 6.3 (1988), pp. 296–339.

[30] Committee, Alpha Architecture et al. *Alpha architecture reference manual*. Digital Press, 2014.

[31] *Common Vulnerabilities and Exposures*. 2019. URL: https://cve.mitre.org/ (visited on 15/05/2019).

[32] Costan, Victor and Devadas, Srinivas. 'Intel SGX Explained.' In: *IACR Cryptology ePrint Archive* 2016.086 (2016), pp. 1–118.

[33] Costan, Victor, Lebedev, Ilia and Devadas, Srinivas. 'Sanctum: Minimal hardware extensions for strong software isolation'. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 857–874.

[34] *Coyotos Microkernel Specification*. URL: https://web.archive.org/web/20160904092954/http://www.coyotos.org:80/docs/ukernel/spec.html (visited on 28/05/2020).

[35] *CVE - CVE-2019-17349*. 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17349 (visited on 20/11/2019).

[36] Dang, Thurston HY, Maniatis, Petros and Wagner, David. 'The performance cost of shadow stacks and stack canaries'. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM. 2015, pp. 555–566.

[37] Davis, Brooks et al. 'CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2019, pp. 379–393.

[38] Dees, Brian. 'Native command queuing-advanced performance in desktop storage'. In: *IEEE Potentials* 24.4 (2005), pp. 4–7.

[39] Dennis, Jack B and Van Horn, Earl C. 'Programming semantics for multiprogrammed computations'. In: *Communications of the ACM* 9.3 (1966), pp. 143–155.

[40] Devices, A Micro. *AMD64 architecture programmer's manual volume 2: System programming*. 2006.

[41] Dhawan, Udit et al. 'Architectural support for software-defined metadata processing'. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 1. ACM. 2015, pp. 487–502.

[42] Dhawan, Udit et al. 'Pump: a programmable unit for metadata processing'. In: *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. ACM. 2014, p. 8.

[43] Dietz, Will and Adve, Vikram. 'Software multiplexing: share your libraries and statically link them too'. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 154.

[44] Du, Zhao-Hui et al. 'Secure encrypted virtualization is unsecure'. In: *arXiv preprint arXiv:1712.05090* (2017).

[45] Dunkels, Adam. 'Design and Implementation of the lwIP TCP/IP Stack'. In: *Swedish Institute of Computer Science* 2 (2001), p. 77.

[46] Durumeric, Zakir et al. 'The matter of heartbleed'. In: *Proceedings of the 2014 conference on internet measurement conference*. ACM. 2014, pp. 475–488.

[47] Ehrenfeld, Jesse M. 'Wannacry, cybersecurity and health information technology: A time to act'. In: *Journal of medical systems* 41.7 (2017), p. 104.

[48] Ferguson, Justin N. 'Understanding the heap by breaking it'. In: *black Hat USA* (2007), pp. 1–39.

[49] Filardo, Nathaniel Wesley et al. 'Cornucopia'. In: 2020.

[50] Fillo, Marco et al. 'The m-machine multicomputer'. In: *International Journal of Parallel Programming* 25.3 (1997), pp. 183–212.

[51] Gibbs, Samuel. 'Chinese Android Phones Contain In-Built Hacker 'Backdoor''. en-GB. In: *The Guardian* (Dec. 2014).

[52] Golub, David B et al. 'UNIX as an Application Program.' In: *UsENIX summer*. 1990, pp. 87–95.

[53] Götzfried, Johannes et al. 'Cache attacks on Intel SGX'. In: *Proceedings of the 10th European Workshop on Systems Security*. ACM. 2017, p. 2.

[54] Gretton-Dann, Matthew. *Arm A-Profile Architecture Developments 2018: Armv8.5-A*. 2018. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a (visited on 27/09/2019).

[55] Grisenthwaite, Richard. *A Safer Digital Future, By Design*. URL: https://www.arm.com/blogs/blueprint/digital-security-by-design (visited on 09/12/2019).

[56] Gruss, Daniel, Maurice, Clémentine and Mangard, Stefan. 'Rowhammer. js: A remote software-induced fault attack in javascript'. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 300–321.

[57] Gueron, Shay. 'Memory encryption for general-purpose processors'. In: *IEEE Security & Privacy* 14.6 (2016), pp. 54–62.

[58] Guide, Part. 'Intel® 64 and ia-32 architectures software developer's manual'. In: *Volume 3B: System programming Guide, Part* 2 (2011).

[59] Hardy, Norm. 'The Confused Deputy: (or Why Capabilities Might Have Been Invented)'. In: *SIGOPS Oper. Syst. Rev.* 22.4 (Oct. 1988), pp. 36–38. URL: http://doi.acm.org/10.1145/54289.871709.

[60] Hardy, Norman. 'KeyKOS architecture'. In: *ACM SIGOPS Operating Systems Review* 19.4 (1985), pp. 8–25.

[61] Houdek, Merle E, Soltis, Frank G and Hoffman, Roy L. 'IBM System/38 support for capability-based addressing'. In: *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press. 1981, pp. 341–348.

[62] Hunt, Galen C and Larus, James R. 'Singularity: rethinking the software stack'. In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49.

[63] *IPC Performance of seL4 microkernel on RISC-V Platforms*. 2018. URL: http://heshamelmatary.blogspot.com/2018/06/ipc-perfoamnce-of-sel4-microkernel-on.html (visited on 01/10/2019).

[64] Irazoqui, Gorka et al. 'Wait a minute! A fast, Cross-VM attack on AES'. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2014, pp. 299–319.

[65] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, June 2018, p. 520. URL: https://www.iso.org/standard/74528.html.

[66] Jim, Trevor et al. 'Cyclone: A Safe Dialect of C.' In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[67] Joannou, Alexandre et al. 'Efficient tagged memory'. In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 641–648.

[68] Jones, Richard WM and Kelly, Paul HJ. 'Backwards-compatible bounds checking for arrays and pointers in C programs'. In: *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*. 001. Linköping University Electronic Press. 1997, pp. 13–26.

[69] K, Paul. *Replicant Developers Find and Close Samsung Galaxy Backdoor — Free Software Foundation — Working Together for Free Software*. Mar. 2014. URL: https://www.fsf.org/blogs/community/replicant-developers-find-and-close-samsung-galaxy-backdoor.

[70] Kaplan, David, Powell, Jeremy and Woller, Tom. 'AMD memory encryption'. In: *White paper* (2016).

[71] Kim, Yoongu et al. 'Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors'. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, pp. 361–372.

[72] Klein, Gerwin et al. 'seL4: Formal verification of an OS kernel'. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[73] Kocher, Paul et al. 'Spectre attacks: Exploiting speculative execution'. In: *arXiv preprint arXiv:1801.01203* (2018).

[74] Kuszmaul, Bradley C. 'SuperMalloc: a super fast multithreaded malloc for 64-bit machines'. In: *Proceedings of the 2015 International Symposium on Memory Management*. 2015, pp. 41–55.

[75] Kuznetsov, Volodymyr et al. 'Code-pointer integrity'. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 147–163.

[76] Kwon, Albert et al. 'Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security'. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 721–732.

[77] Lattner, Chris and Adve, Vikram. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[78] Levy, Henry M. *Capability-based computer systems*. Digital Press, 2014.

[79] Liedtke, Jochen. 'Toward real microkernels'. In: *Communications of the ACM* 39.9 (1996), pp. 70–78.

[80] Liétar, Paul et al. 'snmalloc: a message passing allocator'. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. ACM. 2019, pp. 122–135.

[81] Lipp, Moritz et al. 'Armageddon: Cache attacks on mobile devices'. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 549–564.

[82] Lipp, Moritz et al. 'Meltdown'. In: *arXiv preprint arXiv:1801.01207* (2018).

[83] *LLVM Website*. URL: https://llvm.org/ (visited on 12/09/2019).

[84] Madhavapeddy, Anil and Scott, David J. 'Unikernels: the rise of the virtual library operating system'. In: *Communications of the ACM* 57.1 (2014), pp. 61–69.

[85] Marinos, Ilias, Watson, Robert NM and Handley, Mark. 'Network stack specialization for performance'. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 175–186.

[86] Markettos, A Theodore et al. 'Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals.' In: *NDSS*. 2019.

[87] Martins, D. Okoi. *The 7 Most Popular Programming Languages on GitHub in 2019*. 30th Apr. 2019. URL: https://www.fossmint.com/popular-programming-languages-on-github/ (visited on 04/11/2019).

[88] Mayer, Alastair JW. 'The architecture of the Burroughs B5000: 20 years later and still ahead of the times?' In: *ACM SIGARCH Computer Architecture News* 10.4 (1982), pp. 3–10.

[89] Miller, Mark S, Yee, Ka-Ping, Shapiro, Jonathan et al. *Capability myths demolished*. Tech. rep. Technical Report SRL2003-02, Johns Hopkins University Systems Research ..., 2003.

[90] *MINIX 3*. URL: https://wiki.minix3.org/doku.php?id=www:documentation:read-more (visited on 23/10/2019).

[91] Morbitzer, Mathias et al. 'Severed: Subverting amd's virtual machine encryption'. In: *Proceedings of the 11th European Workshop on Systems Security*. 2018, pp. 1–6.

[92] N.M. Watson, Robert. *The Arm Morello Board*. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html (visited on 09/12/2019).

[93] Nagarakatte, Santosh et al. 'CETS: compiler enforced temporal safety for C'. In: *ACM Sigplan Notices*. Vol. 45. 8. ACM. 2010, pp. 31–40.

[94] Nagarakatte, Santosh et al. 'SoftBound: Highly compatible and complete spatial memory safety for C'. In: *ACM Sigplan Notices* 44.6 (2009), pp. 245–258.

[95] Nahum, Erich, Barzilai, Tsipora and Kandlur, Dilip. 'Performance issues in WWW servers'. In: *ACM SIGMETRICS Performance Evaluation Review* 27.1 (1999), pp. 216–217.

[96] Needham, Roger M and Walker, RD Ho. 'The Cambridge CAP computer and its protection system'. In: *ACM SIGOPS Operating Systems Review* 11.5 (1977), pp. 1–10.

[97] Nethercote, Nicholas and Seward, Julian. 'Valgrind: a framework for heavyweight dynamic binary instrumentation'. In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[98] Newzoo. *Number of smartphone users worldwide from 2016 to 2021 (in billions) [Graph]*. 2018. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 16/10/2019).

[99] Ngabonziza, Bernard et al. 'Trustzone explained: Architectural features and use cases'. In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2016, pp. 445–451.

[100] Oleksenko, Oleksii et al. 'Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches'. In: *arXiv preprint arXiv:1702.00719* (2017).

[101] Oracle. *Hypervisor and Logical Domains - Oracle VM Server for SPARC 2.0 Administration Guide*. 2010. URL: https://docs.oracle.com/cd/E19608-01/html/821-1485/hypervisorandldoms.html (visited on 11/11/2019).

[102] Pai, Vivek S, Druschel, Peter and Zwaenepoel, Willy. 'IO-Lite: a unified I/O buffering and caching system'. In: *ACM Transactions on Computer Systems (TOCS)* 18.1 (2000), pp. 37–66.

[103] Peter, Simon et al. 'Arrakis: The operating system is the control plane'. In: *ACM Transactions on Computer Systems (TOCS)* 33.4 (2015), pp. 1–30.

[104] Peter, Simon et al. 'Early experience with the Barrelfish OS and the Single-Chip Cloud Computer.' In: *MARC Symposium*. 2011, pp. 35–39.

[105] Pike, Rob. 'The go programming language'. In: *Talk given at Google's Tech Talks* (2009).

[106] Rashid, Richard F and Tokuda, Hideyuki. 'Mach: a system software kernel'. In: *Computing systems in engineering* 1.2-4 (1990), pp. 163–169.

[107] Redell, David D. *Naming and protection in extendible operating systems*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1974.

[108] Richardson, Alexander. *Complete spatial safety for C and C++ using CHERI capabilities*. Tech. rep. UCAM-CL-TR-949. University of Cambridge, Computer Laboratory, June 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf.

[109] Roessler, Nick and DeHon, André. 'Protecting the stack with metadata policies and tagged hardware'. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 478–495.

[110] Sabbagh, Dan. 'Calls for Backdoor Access to WhatsApp as Five Eyes Nations Meet'. en-GB. In: *The Guardian* (July 2019).

[111] Sasaki, Hiroshi et al. 'Practical Byte-Granular Memory Blacklisting using Califorms'. In: *arXiv preprint arXiv:1906.01838* (2019).

[112] Schrammel, David et al. 'Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86'. In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Association. 2020.

[113] Schüpbach, Adrian et al. 'Embracing diversity in the Barrelfish manycore operating system'. In: *Proceedings of the Workshop on Managed Many-Core Systems*. Vol. 27. 2008.

[114] Seaborn, Mark and Dullien, Thomas. 'Exploiting the DRAM rowhammer bug to gain kernel privileges'. In: *Black Hat* 15 (2015).

[115] *Sel4 Performance*. 2019. URL: https://sel4.systems/About/Performance/home.pml (visited on 01/10/2019).

[116] *seL4 Reference Manual*. 2019. URL: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf (visited on 17/05/2021).

[117] Serebryany, Konstantin et al. 'AddressSanitizer: A Fast Address Sanity Checker'. In: Presented as Part of the 2012 {USENIX Annual Technical Conference ({USENIX {ATC 12). 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany (visited on 04/11/2019).

[118] Shapiro, Jonathan S, Smith, Jonathan M and Farber, David J. *EROS: a fast capability system*. Vol. 33. 5. ACM, 1999.

[119] Singhania, Akhilesh et al. *Capability Management in Barrelfish*. Tech. rep. ETH, Zurich, 2017. URL: http://www.barrelfish.org/publications/TN-013-CapabilityManagement.pdf.

[120] Skorstengaard, Lau, Devriese, Dominique and Birkedal, Lars. 'Reasoning about a machine with local capabilities'. In: *European Symposium on Programming*. Springer. 2018, pp. 475–501.

[121] Skorstengaard, Lau, Devriese, Dominique and Birkedal, Lars. 'StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities'. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), p. 19.

[122] Soghoian, Christopher. 'Caught in the cloud: Privacy, encryption, and government back doors in the web 2.0 era'. In: *J. on Telecomm. & High Tech. L.* 8 (2010), p. 359.

[123] Song, Dokyung et al. 'SoK: sanitizing for security'. In: *arXiv preprint arXiv:1806.04355* (2018).

[124] Stepanov, Evgeniy and Serebryany, Konstantin. 'MemorySanitizer: fast detector of uninitialized memory use in C++'. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2015, pp. 46–55.

[125] Stoep, Jeff Vander. *Android: protecting the kernel*. 2016. URL: https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf (visited on 30/08/2019).

[126] Suzaki, Kuniyasu et al. 'Memory deduplication as a threat to the guest OS'. In: *Proceedings of the Fourth European Workshop on System Security*. 2011, pp. 1–6.

[127] Suzaki, Kuniyasu et al. 'Software side channel attack on memory deduplication'. In: *ACM Symposium on Operating Systems Principles (SOSP 2011), Poster session*. 2011.

[128] Tsampas, Stelios, Devriese, Dominique and Piessens, Frank. 'Temporal safety for stack allocated memory on capability machines'. In: *IEEE Computer Society Press* (2019).

[129] Van Bulck, Jo et al. 'Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution'. In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 991–1008.

[130] Waldspurger, Carl A. 'Memory resource management in VMware ESX server'. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 181–194.

[131] Watson, Robert N. M. et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Tech. rep. UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, June 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf.

[132] Watson, Robert NM et al. 'Capsicum: Practical Capabilities for UNIX.' In: *USENIX Security Symposium*. Vol. 46. 2010, p. 2.

[133] Weisse, Ofir, Bertacco, Valeria and Austin, Todd. 'Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves'. In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 81–93.

[134] Wilkes, Maurice Vincent and Needham, Roger Michael. 'The Cambridge CAP computer and its operating system'. In: (1979).

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

[136] Woodruff, Jonathan et al. 'The CHERI Capability Model: Revisiting RISC in an Age of Risk'. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468. URL: http://dl.acm.org/citation.cfm?id=2665671.2665740 (visited on 03/02/2016).

[137] Wulf, William et al. 'Hydra: The kernel of a multiprocessor operating system'. In: *Communications of the ACM* 17.6 (1974), pp. 337–345.

[138] Xia, Hongyan et al. 'CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety'. In: (2019).

[139] Xiao, Jidong et al. 'Security implications of memory deduplication in a virtualized environment'. In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2013, pp. 1–12.

[140] Yang, Xi et al. 'Why nothing matters: the impact of zeroing'. In: *Acm Sigplan Notices*. Vol. 46. 10. ACM. 2011, pp. 307–324.

[141] Zhang, Tong, Lee, Dongyoon and Jung, Changhee. 'BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free'. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2019, pp. 631–644.