Technical Report

UCAM-CL-TR-976
ISSN 1476-2986

Number 976

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

CHERI compartmentalisation for embedded systems

Hesham Almatary

November 2022

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

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

This technical report is based on a dissertation submitted February 2022 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St John’s College.

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

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

ISSN 1476-2986

Abstract

CHERI Compartmentalisation for Embedded Systems

Hesham Moustafa Khaled Almatary

Embedded system designers are facing an inexorable pressure to add more features and leverage connectivity. This creates potential attack vectors in areas that were not subject to security concerns before. Individuals' privacy could be violated, cars and planes could crash, credit-card details could be stolen, and medical devices could critically malfunction, affecting vital life-concerning tasks or leaking sensitive patients' details. *Software compartmentalisation* has the potential to manage the attack vector better, defend against unknown future software vulnerabilities, and limit the consequences of potential successful attacks to the compromised component without affecting the rest of the system. Unfortunately, the current state-of-the-art security technologies for embedded systems (e.g., MPUs) are not well-designed for implementing fine-grained software compartmentalisation while meeting embedded systems requirements. They suffer from inherent design issues that limit scalability, compatibility, security, and performance.

This dissertation proposes *CompartOS* as a new lightweight hardware-software compartmentalisation model building on CHERI (a hardware capability architecture) to secure mainstream and complex embedded software systems. CompartOS is an automatic, linkage-based compartmentalisation model that isolates mutually distrusting linkage modules (e.g., third-party libraries) executing in a single-address-space and single-privilege-ring environment. Further, CompartOS enables the management of faults within software components by introducing support for partial recovery, thus improving availability while maintaining compatibility by requiring minimal development efforts—a critical requirement for many embedded systems.

We have implemented multiple prototypes of compartmentalisation models, including MPU-based protection and CompartOS, in FreeRTOS and compared them in performance, compatibility, security, and availability. Microbenchmarks show that CompartOS' protection-domain crossing is 95% faster than MPU-based IPC. We applied the CompartOS model, with low effort, to complex, mainstream systems, including TCP servers, Amazon's OTA updates, and a safety-critical automotive demo. CompartOS not only catches 10 out of 13 FreeRTOS-TCP published vulnerabilities that MPU-based protection (e.g., uVisor) cannot catch but can also recover from them, maintaining the availability of safety-critical systems. Further, our TCP throughput evaluations show that our CompartOS prototype is 52% faster than the most relevant and advanced MPU-based compartmentalisation model (e.g., ACES), with a 15% overhead compared to an unprotected system. This comes at an FPGA's LUTs overhead of 10.4% to support CHERI for an unprotected baseline RISC-V processor, compared to 7.6% to support MPU, while CHERI only incurs 1.3% of the registers area overhead compared to 2% for MPU.

ACKNOWLEDGEMENTS

I would like to dedicate the time and efforts I have spent during the course of my PhD to my family; I would not be where I am today if it was not for them. My father and my role model has always supported and pushed me to be a better version of myself. I have been, and always will be, learning from and looking up to him. My mother, who is the tenderness of my heart, has provided ease during trying moments through her laughter and optimism. My twin sister and best friend has always been there when I needed her, and with whom I have shared life-long experiences and memories. Thank you. I could not have asked for a better family to call my own.

Special thanks to my supervisor, Robert N. M. Watson, for giving me the opportunity to work on a research topic which reflects my passion. Robert has always been understanding, patient, and supportive. They say a good supervisor is half of a PhD, and I have been lucky to have a great one. Michael Dodson has been a great colleague and a friend. He has always motivated and pushed me through our detailed discussions and collaborations, fueled by his enthusiasm. Despite his own demanding schedule, he continuously helped and supported me. Thanks, Mike!

Working in a brilliant research group, I have been fortunate to interact with and build on the outstanding efforts of several of my colleagues that made the CHERI project what it is today. Thanks to Simon Moore, Peter Sewell, Peter Neumann, and Brooks Davis for overseeing the project and providing suggestions and feedback throughout my PhD. Further, I am grateful to my colleagues: Alexander Richardson, Jessica Clarke, Peter Rugg, Alexandre Joannou, Jonathan Woodroof, Nathaniel W. Filardo, George Neville-Neil, Hongyan Xia, Lawrence Esswood, Alfredo Mazzinghi, Brett Gutstein, A. Theo Markettos, Ivan Ribeiro, and Franz Fuchs. I have built on their efforts and help, and my work would not have been possible without theirs.

Finally, I would like to thank my examiners, Alastair Beresford and Hugo Vincent, for putting so much effort into reviewing my thesis, and discussing it with me during the viva. Your feedback has been extremely helpful, and I also enjoyed the valuable discussions about my research. Thanks for making the viva go so smoothly; it was a fun and memorable experience!

Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under contracts HR0011-18-C- 0016 (“ECATS”) and HR0011-18-C-0013 (“BESSPIN”). The views, opinions, and/or findings contained in this dissertation are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Contents

GLOSSARY

ACL Access Control List.

API Application Programming Interface.

AWS Amazon Web Services.

CHERI Capability Hardware Enhanced RISC Instructions.

DoS Denial of Service.

EOS Embedded Operating System.

FPGA Field Programmable Gate Arrays.

IoT Internet of Things.

IPC Inter Process Communication.

LLVM Low Level Virtual Machine.

LUT Look-Up Table.

MMU Memory Managment Unit.

MPU Memory Protection Unit.

OS Operating System.

OTA Over The Air.

PMP Physical Memory Protection.

ROP Return Oriented Programming.

RTOS Real Time Operating System.

TCB Trusted Computing Base.

TLB Translation Lookaside Buffer.

TLS Thread Local Storage.

CHAPTER 1

INTRODUCTION

The world is getting more connected every day, with embedded devices and the Internet of Things (IoT) becoming essentials. Connected devices are deployed in smart homes, toys, wearables, automotive, security cameras, healthcare, and avionics. The inexorable need to add more features and leverage connectivity creates potential attack vectors [1–8] in areas that were not subject to security concerns before. Individual's privacy could now be violated, cars and planes could crash, credit-card details could be stolen, and medical devices could critically malfunction, affecting vital life-concerning actions or leak sensitive patients' details. The prevalence of such devices creates large scale risks for the economy, national security, and the safety of large populations.

Broadly speaking, attacks on embedded systems can be categorised into hardware, communication channels, and vulnerability-based software attacks. Hardware attacks require having physical access to the device. Based on the security analysis of embedded systems in [9–11], this includes physical tampering, reverse engineering, and side-channel attacks. Communication-channel attacks include eavesdropping on the network to try to steal private information such as secret keys or privacy-related data. Other communication channel attacks may attempt to manipulate the integrity of data to create a desired, malicious outcome or the availability of data in a Denial of Service (DoS). Finally, vulnerability-based software attacks include cryptography errors, software protocols abuse, authentication and authorisation (e.g., extracting passwords), and finally exploiting software vulnerabilities (e.g., buffer overflows) to achieve remote code executing, DoS, or confidentiality-related attacks.

Embedded systems and IoT get more complex and feature-rich every day. While complexity increases, they tend to maintain the real-time requirements and determinism of previous generations. We are narrowing down the scope of systems we are targetting, to the mainstream, large, feature-rich systems that are unprotected and require some form of security to defend against unknown and future vulnerability-based software attacks. That is, we do not try to provide a security model or solution for small embedded systems that get created from scratch. To protect such systems from the inevitable vulnerabilities exploitable in the software, all protection models require some hardware support, such as privilege rings, Memory Protection Unit (MPU)s, TrustZone [12], or Memory Managment Unit (MMU)s. However, there is currently a gap in hardware-software security for targetted embedded applications; the M-class-like processors [13] with MPU and TrustZone are too small and inadequate for their needs in terms of scalability, memory requirements, and fine-grained security, while A-class-like processors [14] and MMUs do not meet the fine-grained security or real-time and determinism requirements. Thus, such systems tend to stay unprotected or apply complicated workarounds and manual task-based compartmentalisation to get some form of security with non-trivial development, maintenance, and time-to-market overheads. The scope such systems fall in is safety-critical avionics and automotive. Examples of deployed systems in safety or privacy-critical areas that either deploy no security or are forced to compromise with complex security solutions are:

- Amazon's FreeRTOS with WiFi, TCP/IP, and Bluetooth stacks running on high-end M-class processors (Amazon Web Services (AWS) Reference Integrations [15]) used for feature-rich IoT.
- Primus Epic Avionics: Deos and ARINC 653, running on x86 and Arm A-class [16].
- VxWorks CERT EDITION, running on NXP QorIQ for automotive and avionics [17].
- RTEMS, used in NASA's Magnetosphere Multiscale (MMS) Mission [18, 19] running on the Coldfire CPU [20].

There have been multiple recent attempts [21] to provide software compartmentalisation, sandboxing, and isolation in embedded systems by relying on MPUs as a standard hardware feature for embedded systems. Due to the hardware limitations of MPUs (by design) [22], such solutions are not scalable and do not provide fine-grained protection, especially for rich and complex mainstream software stacks with hundreds of thousands of lines of code (LoC) and many resources. For example, the state-of-the-art deployed Real Time Operating System (RTOS)es such as FreeRTOS-MPU [23], TockOS [24], and Mbed uVisor [25] rely on an MPU-based task or process model for isolation. This could work fine for relatively small embedded systems with a small number of resources. However, for mainstream applications to be secured using such technologies, they will have to be ported to use new Application Programming Interface (API)s or memory-safe languages (e.g., Rust) and redesigned around tasks or processes. This could be a deal-breaker for large and mainstream projects that are written in different languages and not designed around tasks or processes due to the time-to-market requirements and development overheads. Further, MPU hardware scales poorly with tens or hundreds of compartments and resources that need to be isolated. ACES [26] tries to provide an automatic compartmentalisation technique that is source-file or IO-based for small embedded systems, using the MPU. This helps with the compatibility requirement by not requiring to port or refactor mainstream code. While promising for small and simple embedded systems, ACES still cannot target large software projects due to the MPU hardware limitations. Further, ACES does not try to provide an Operating System (OS) protection model, including threading, secure interrupt handling, or dynamic memory allocation. TrustLite [27] and TyTan [28] try to mitigate the MPU design limitations by increasing the number of memory regions to protect. However, increasing the MPU regions does not scale in either hardware due to the increased associative lookups or in software as context switches will incur O(N) performance overhead where N is the number of memory regions to protect, per protection domain. In almost all of the MPU-based protection published literature, the limitations of MPU scalability are admitted, and, therefore, they only try to target small systems. Further, none of these works attempts to address availability, an essential requirement of safety-critical systems. This leaves large, mainstream and complex embedded systems that require some form of security while maintaining determinism as a real-time requirement still vulnerable.

CompartOS aims to fill this security requirement gap in mainstream, complex, high-end embedded systems, where manually porting them to use MPU or MMU hardware, OS' APIs, and/or rewriting them in memory-safe languages is inadequate or requires significant redesign and reimplementation efforts which would have a high development, testing, and deployment cost and are error-prone. More specifically, CompartOS aims to reduce the effects of programming errors and vulnerabilities between software components running on the same embedded processor, in the same address space and privilege ring and sharing resources by applying a novel software compartmentalisation technique that builds on Capability Hardware Enhanced RISC Instructions (CHERI). Compatibility, availability, and scalability are the main design goals in this work.

The primary research contribution of this thesis is the CompartOS model (and its evaluation via prototyping) as a linkage-based, automatic compartmentalisation model for embedded systems. Building on CHERI to provide software compartmentalisation protects against memory-safety attacks that represent 70% of software vulnerabilities in commodity OSes according to Microsoft [29]. However, in embedded systems, it is not enough to catch a security violation but to appropriately handle it to maintain the integrity and availability of security requirements. In terms of contributions, CompartOS, as a model for CHERI-enabled, MMU-less, embedded systems, is comparable to the well-defined UNIX process model (inspired by Multics [30, 31]) for general-purpose systems.

CompartOS is novel and differs from state-of-the-art systems. First, CompartOS significantly differs from previous CHERI work as follows:

- Unlike [32, 33], CompartOS focuses on embedded systems by enforcing complete protection and compartmentalisation solely using CHERI (i.e., by replacing MMUs and MPUs) instead of being complementary to the MMU-based UNIX process model.

- Unlike CheriOS [34] and CheriRTOS [35] (where both are task-based), CompartOS does not introduce a new OS or API to manually compartmentalise complex software, but it automatically enforces compartmentalisation based on the linkage model and programming languages.

- Unlike [32, 33], CompartOS leverages system-level compartmentalisation, rather than just application-level compartmentalisation, on all OS aspects, including kernels, device drivers, OS libraries, and applications.

- Focuses on availability and recovery in safety-critically embedded systems after catching CHERI faults (or others), rather than accepting fail-stop on, for example, a memory protection fault.

Second, unlike MPU-based state-of-the-art secure embedded architectures [21, 24–28, 36], CompartOS:

- Scales to hundreds of compartments and resources without hardware protection limitations.

- Targets existing mainstream and complex unprotected embedded (operating) systems and libraries that could scale to millions of lines of code.

- Provides capability-based security that is enforced on every pointer and linkage module, inter- and intra- task/application.

- Assumes malicious compartments and aims to protect against current and unknown future software vulnerabilities.

- Outperforms comparable state-of-the-art systems.

In summary, we describe the major contributions in this dissertation as follows:

CompartOS as an abstract model A generic CHERI-based compartmentalisation and protection model for embedded (operating) systems, which can be effortlessly applied to most mainstream embedded systems and RTOSes.

CheriFreeRTOS as an application of the CompartOS model A real-world, sound prototype of the CompartOS model in FreeRTOS including a secure loader and protecting the kernel, OS/IoT libraries, and large and complex deployed mainstream applications.

MPU-based comparisons Implementations of the state-of-the-art MPU-based protection models in FreeRTOS to fairly compare the CheriFreeRTOS prototype against, at scale.

Evaluation Demonstrating complete and functional real-world, deployed use cases with CompartOS and other MPU-based models and providing thorough security, compatibility, recovery, and performance analysis. We also report the lessons learned from that.

This dissertation is outlined as follows. Chapter 2 gives a background on embedded systems and security and specifically the scope that CompartOS is targetting. Software compartmentalisation is also motivated with examples and mechanisms within the state-of-the-art general-purpose and embedded systems. Further, we provide an overview of CHERI and its software protection models. Chapter 3 introduces the CompartOS model as the first contribution of this dissertation. It discusses the embedded systems security requirements before stating the design goals of CompartOS and how they are met. In the same chapter, we discuss a few fault handling and recovery mechanisms, part of the model to demonstrate its flexibility and the focus on availability. Chapter 4 describes CheriFreeRTOS as an example application of the CompartOS model along with a secure compartment loader. Chapter 5 discusses the last two contributions—MPU-based implementations and evaluation. The chapter starts by measuring the hardware costs of adding MPU, MMU, and CHERI to a baseline RISC-V processor. After that, it evaluates different software protection prototypes in FreeRTOS, including CompartOS, CHERI, and MPU-based protection across multiple benchmarks. At the end of the chapter, we provide complete use cases with real-world attacks and fault handling scenarios within mainstream, complex, and safety-critical demonstrators. Chapter 6 discusses related work on MPU-based and CHERI-based protection.

CHAPTER 2

Background

In this chapter, we describe earlier work on which CompartOS builds, from mainstream embedded system hardware and software stacks to CHERI and MPU support for memory protection and compartmentalisation.

Section 2.1 narrows down the scope of embedded systems we are targetting, while 2.2 gets into more details about traditional embedded operating systems that we are concerned about. Section 2.3 discusses security in embedded systems, including the type of attacks and approaches to defend against them using off-the-shelf hardware protection units. Section 2.5 defines what software compartmentalisation means and how it could help defend against a good range of attacks in embedded systems. It also provides new terminology that we use across this dissertation. In the same section, we discuss state-of-the-art MPU-based compartmentalisation approaches, which are used to offer security for embedded systems. Section 2.6 discusses existing fault handling techniques that are commonly used after catching integrity and confidentiality violations. In the remainder of the chapter, we discuss capability-based systems and their potential to greatly enhance embedded systems security. Finally, we give an overview of CHERI as a capability-based hardware architecture that we build on to develop our new CompartOS software compartmentalisation model.

2.1 Embedded Systems

Embedded systems represent a field that has diverse requirements and target applications [37] compared to general-purpose systems (e.g., laptops or time-shared personal computers). The latter focuses on multi-user and time-sharing requirements and is regarded to be general-purpose enough to meet most of the daily tasks a person might need for work or personal activities. The goal of general-purpose systems is to protect users and programs from one another while providing them with a smooth interaction experience. On the other hand, embedded systems are customised to perform specific tasks and are more diverse; they are mostly single-user and designed around applications and libraries working in a single-address-space environment.

While CompartOS can be applied to mainstream baremetal and tiny embedded systems (at a cost), it aims to secure the high-end range of *mainstream* embedded operating systems (including RTOSes). By mainstream, we mean traditional embedded (operating) systems that are mostly unprotected and shy away from using MPUs and MMUs as they do not meet their requirements. This is in contrast to using *new* security architectures that target small systems (e.g., TockOS and ACES) and require existing applications to be (re)written on top of them.

	MCU	**Low-end**	**High-end**
Memory Subsystems	Optional caches. TCM, SRAM and ROM ($<$ 2 MiB), No DRAM.	Small caches, ROM, DRAM ($>$ 16 MiB)	Large caches ($>$ 1 MiB)). Large DRAM ($>$ 1 GiB)
Processor Pipeline	in-order, 3-5 stages	in-order or out-of-order, 5-8 stages	out-of-order $>$ 8 stages
Hardware Protection	Optional MPU and TrustZone	MPU, MMU, and TrustZone	MPU, MMU, TrustZone, Hypervisor ring
Example Boards	RISC-V's HiFive1 [38], FRDM-K64F [39]	BeagleBoneBlack [40]	M1 Chips [41], HiFive Unleashed [42]
Requirements	Low-power, small memory footprints	Low-cost, determinism, relatively large memory	Virtualisation, multicore, High-performance
Example OSes	TockOS, Mbed	FreeROS, RTEMS, VxWorks	Android, (Embedded) Linux, FreeBSD
Domains	MCUs, tiny, power-conservative devices	Avionics, automotive, smart homes, robotics	Personal computers, servers

Table 2.1: Embedded systems categorisations in the MCU and low-end columns compared to general-purpose category in the high-end column. CompartOS targets low-end systems.

Traditional embedded operating systems are facing security concerns and need some form of protection while continuing to meet their real-time and safety-critical requirements such as partitioning, bounded processing, high determinism and high throughput. Developers either try to use the MPU in MCUs that provide it, but that is not scalable or fine-grained enough for complex applications, or they go for general-purpose (e.g., A-class Arm processors) that have MMUs (that are not frequently used due to complexity and performance non-determinism) and rather use the MPU. The area those systems fit in is safety-critical avionics and automotive. For example, FreeRTOS or RTEMS run on Raspberry Pi [43] or Beagle [44] boards (embedded with Arm A-class processors) without an MMU-based process or protection model. We categorise computer systems as in Table 2.1. CompartOS mainly targets the low-end A-class category. Example deployed systems are:

- Amazon's FreeRTOS with WiFi, TCP/IP, and Bluetooth stacks running on high-end M-class processors (AWS Reference Integrations [15]) and used for feature-rich IoT.
- Primus Epic Avionics: Deos and ARINC 653, running on x86 and Arm's A-class [16].
- VxWorks CERT EDITION, running on NXP QorIQ [45] for automotive and avionics [17].
- RTEMS used in NASA's Magnetosphere Multiscale (MMS) Mission [18, 19] running on the Coldfire CPU [20].

In the following section, we talk in more detail about traditional embedded operating systems (EOSes) that CompartOS mainly targets and discuss the common design and implementation decisions they share.

2.2 Embedded Operating Systems

Embedded operating systems (including RTOSes) are more advanced than baremetal systems by mainly providing threading and scheduling support. They differ from general-purpose OSes (e.g., Linux) in that they are simple, application-oriented (rather than users and groups oriented), relatively smaller and less complex in terms of the services they provide. For instance, typical UNIX-based OSes come with pre-existing complexity having to manage virtual memory, system calls, filesystems, and networking, built into the kernel. In contrast, most embedded systems do not have a process model that supports virtualisation and the illusion that an application has access to the entire contiguous memory. While some embedded systems can provide filesystems and networking, such services are optional and are not considered built-in, part of the core kernel. Figure 2.1 from [37] places traditional embedded OSes (EOS) between baremetal and process-model OSes (PMOS) along with trade-offs.

While CompartOS can be applied to baremetal and PMOS systems, we are mainly targetting high-end traditional embedded OSes for the purpose of this thesis. There are many mainstream

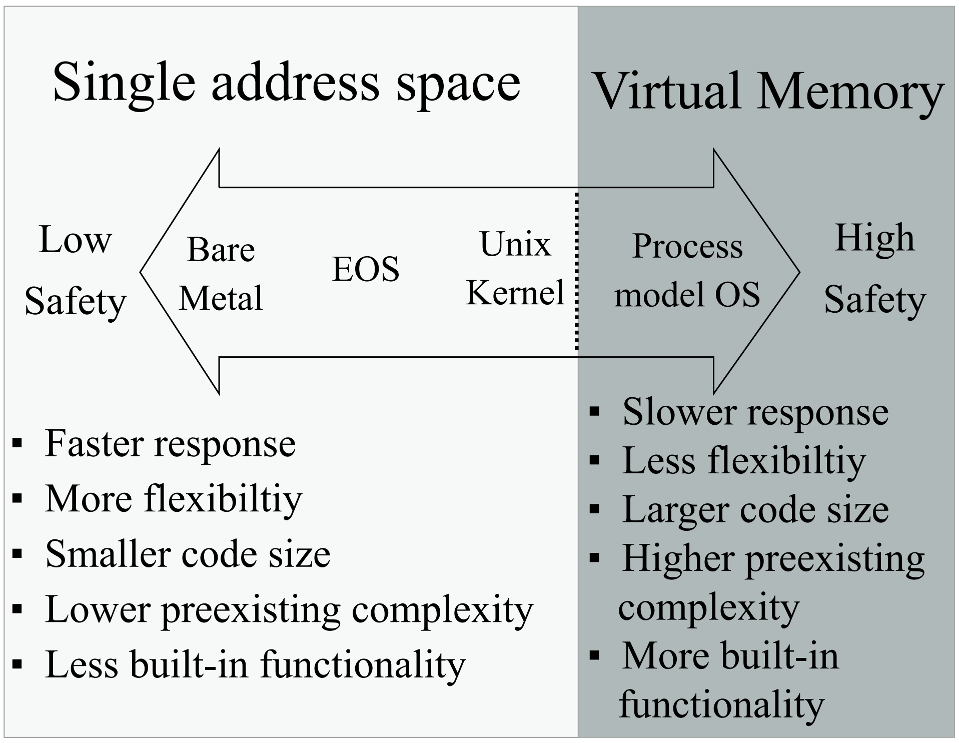


Figure 2.1: Safety versus Speed in Embedded Systems. The figure is drawn with permission from George V. Neville-Neil, the author of [37].

open-source and proprietary OSes of this category that do need to be secured, but the current hardware-software embedded security solutions are not a good fit for them. Example OSes of this category are RTEMS [46], FreeRTOS [47], VxWorks [17], and Deos [16]. They all share common attributes such as executing in a single-address-space, without a process model or virtualisation support and they do not have many constraints on memory footprints, power consumption, and area size. The main requirements are faster response, flexibility, and determinism. Further, most of such EOSes are feature-rich and thus follow a modular software design (to manage complexity and motivate re-usability) where each software subsystem is well-defined with clear boundaries and APIs between them. In the source-code, those subsystems are placed in separate directories, and they are executing as software libraries at runtime. Many of those OSes are mostly libraries-centric rather than task-centric. RTEMS, for example, uses FreeBSD's network stack as a third-party library [48]. Similarly, FreeRTOS uses Arm's MbedTLS library for its secure internet connectively [49]. It is clear that this trend of using third-party libraries (instead of reinventing the wheel and reimplementing them from scratch), each with its different assurance levels, is common for mainstream EOSes that need to keep up with new IoT standards (e.g., MQTT, TLS, cryptography protocols). Integrating such libraries in a single-address-space EOS with no protection is a huge risk as each comes with its own set of vulnerabilities, and the porting process could be error-prone. Thus, there is a need for flexible compartmentalisation

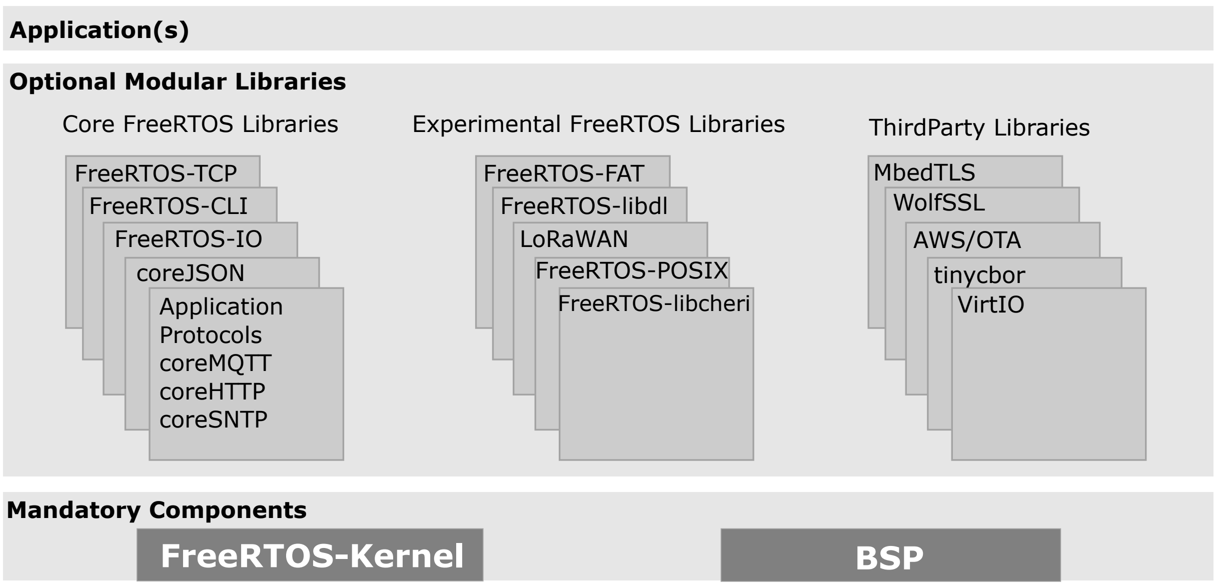


Figure 2.2: Classical FreeRTOS System Layout

that largely maintains the source-code compatibility when adopting such libraries in order to minimise the effects of potential and successful exploits in such third-party libraries.

We have chosen FreeRTOS to represent this category of EOSes as it is open-source, widely used, and has a published set of public vulnerabilities. FreeRTOS is in some sense representative as it targets similar classes of hardware, has similar functionality, and, importantly, has similar vulnerabilities to other EOSes. For example, in Section 5.13, we demonstrate a use case of a mainstream Over The Air (OTA) application that is commonly used in IoT. The demo has about 7 (out of 9) third-party libraries from different vendors.

FreeRTOS makes no assumptions about the presence of hardware protection mechanisms, although it does have limited support for Arm's MPU. FreeRTOS-MPU [23] provides coarse-grained task-based isolation using the MPU by splitting up tasks into privileged and unprivileged, while it only protects the kernel from unprivileged tasks. Developers can choose to build and link a range of FreeRTOS features beyond the central core (FreeRTOS-kernel), including device drivers, a TCP/IP stack, and extensions (e.g., Amazon's OTA update framework [50]), all of which will be linked into a single binary with the application. A classical and modular FreeRTOS software stack is shown in Figure 2.2.

2.3 Security in Embedded Systems

Embedded systems, cyber-physical systems, IoT are all terms used interchangeably to represent low-end, resource-constrained, custom, and real-time devices. They often lack MMU and are small in area to save up manufacturing costs and power consumption. Historically, security was not a concern in that field, but with the recent attacks [1–8] on embedded systems, security started to get traction. Those attacks target the confidentiality, integrity, or availability of the system.

Due to their customised and diverse nature, embedded systems have multiple attack vectors. Broadly speaking, they can be categorised into physical, communication-channel, and vulnerability-based software attacks. Physical attacks require having physical access to the device. Based on the security analysis of embedded systems in [9–11], this includes physical tampering, reverse engineering, and side-channel attacks. Communication-channel attacks include eavesdropping the network to try to steal private information such as secret keys or privacy-related data. In TCP/IP channels, attacks may be in the form of relaying packets. Finally, vulnerability-based software attacks include cryptography errors, software protocols abuse, authentication and authorisation (e.g., extracting passwords), and finally exploiting software vulnerabilities (e.g., buffer overflows) to achieve remote code executing, DoS, or confidentiality-related attacks.

The focus of this thesis is to reduce the effects of programming errors and vulnerabilities between software components running on the same embedded processor and sharing resources by applying a novel software compartmentalisation technique based on CHERI. Compatibility, availability, and scalability are the main design goals in this work.

2.4 Memory Safety

Memory-safety software vulnerabilities have arguably been the most prevalent type of bugs in the history of computers and software in general. In their Eternal War in Memory paper [51], Szekeres et al. argue that memory corruption bugs are one of the oldest problems in computing and still are, regardless of the efforts spent to come up with memory-safe languages and hardware security architectures. Stack and buffer overflows [52] have been one of the most crucial bugs that can be exploited to affect the integrity, confidentiality, and availability of a computer system. Memory safety is thus a vital attack vector in both general-purpose OSes and embedded software systems that are mostly written in memory-unsafe languages such as C/C++ for performance and fine-grained hardware control purposes.

For example, Microsoft has recently revealed [29] that 70% of security software bugs in their systems (e.g., Microsoft Windows written in C/C++) are memory-safety related. This issue does not appear to be only specific to general-purpose OSes such as Windows, but also in the embedded software systems. Papp et al. have done a sound security analysis in embedded systems [9] and found out that memory-safety and programming error attacks on EOSes and firmware are the most frequent and critical.

Some countermeasures against memory safety attacks in embedded systems have been proposed [10]. These include hardware architecture support, static and dynamic analysis tools, compiler support, the use of memory-safe languages, and software compartmentalisation. For example, guard pages [53] rely on the MMU to detect buffer overflows in general-purpose OSes. As embedded OSes do not usually use the MMU, they tend to go for lighter-weight software stack protection solutions. For instance, stack canaries [54] can be optionally enabled by the compiler toolchain to detect stack overflows. Similarly, some EOSes such as RTEMS and FreeRTOS implement their own stack detection in software [55, 56]. However, such solutions are coarse-grained (i.e., only protect the stack of a task, rather than smaller buffers or global objects) and can be bypassed [57].

To overcome some of the overheads (mainly instructions count) and limitations of software stack protection solutions, Armv8.3-A introduced [58] a mechanism called *Pointer Authentication* to defend against some memory safety attacks. In particular, it aims to protect against stack overflow and Return Oriented Programming (ROP)-related attacks. Pointer authentication makes use of the fact that current systems do not make full use of 64-bit addresses, and use some upper bits to hold a Pointer Authentication Code (PAC). With the addition of new instructions to compute and validate PACs using cryptography, address (de)references can be authenticated to enforce security.

Another mechanism that Arm introduced later in Armv8.5-A [59] is Memory Tagging

Extension (MTE). Unlike pointer authentication, MTE does not require instrumentations or source-code modifications; thus, it is claimed to be compatible, scalable, and light-weight. MTE helps to defend against spatial and temporal memory safety attacks by associating a 4-bit tag or colour for every pointer and (16-byte) memory region. Upon accessing a memory region (by the normal load/store instructions), the pointer tag bits are checked by the hardware against the address tag bits. An exception is triggered if they mismatch.

Writing embedded software systems in memory-safe languages such as Rust [60] is also an option. For example, TockOS [24] is written in Rust for tiny embedded systems. However, as most of the existing embedded software systems are already written in C/C++, rewriting them in Rust or porting them to TockOS's API does not maintain the source-code compatibility and could be infeasible for large, mainstream EOSes.

As we discuss later, CHERI [61] is aiming to fix the memory-safety problem starting from the hardware level, to programming languages such as C/C++, up to higher levels in the OS and software compartmentalisation models. While CompartOS is more on the software compartmentalisation spectrum rather than memory-safety, it also entertains the CHERI advantage of protecting pointers and defending against buffer overflows while also being able to recover from other security violations that are not specific to CHERI. This combination of memory-safety and software compartmentalisation countermeasures in CompartOS, in embedded systems, has a great potential to enhance the overall security of embedded systems across the Confidentiality, Integrity, and Availability (CIA) triad.

In the following section, we discuss software compartmentalisation in general, then steer the discussion into software compartmentalisation in embedded systems in particular.

2.5 Software Compartmentalisation

Software compartmentalisation [32, 62–66] is a technique to split up a large monolithic software into smaller compartments in order to reduce the attack vector and limit the effects of a successful attack only to the compromised compartment. Unlike vulnerability mitigation techniques, software compartmentalisation assumes that zero-day unknown vulnerabilities always exist and acts accordingly. The measures that are taken to apply software compartmentalisation follow the *principle of least privilege* [67] by only giving the very minimum privileges to each compartment required to perform a service. This reduces privilege escalation attacks due to the *ambient authority* problem [68] and, thus, maintains the *integrity* and *confidentiality* of the whole system.

Other terms like privilege separation, isolation, partitioning, sandboxing, componentisation, etc., can be thought as sub-categories of compartmentalisation. To be more specific, compartmentalisation targets mainstream, existing insecure monolithic systems and aims to split them up into smaller least-privileged protection domains (compartments) and specify the sharing and communication policies and secure boundaries between them, then enforce "isolation" between them. That is, isolation, partitioning, or sandboxing can be seen as sub-operations of compartmentalisation.

Terms like partitioning and separation often target novel systems that are going to be designed and developed from scratch around new techniques and APIs such as separation kernels (e.g., seL4) that may or may not also provide tools to manage controlled communication and sharing. Thus, compartmentalisation could target both mainstream and new systems.

2.5.1 Computer Protection Terminology

This section discusses general terminology and definitions related to protection in computer systems (within the context of compartmentalisation) along with some objectives we care about.

Protection Domain: An operating system term to define a logical unit of isolation with privileges attached to it. In UNIX-based systems, RTOSes, and microkernels, this has typically been a process, task, or thread. Privileges can be access rights to MMU pages, tables, MPU regions, capability lists, or file descriptor tables. Further, in hardware, there are often privilege rings that separate user applications from the kernel in terms of memory and access to specific hardware features and instructions.

Resources and Privileges: An embedded system has assets that need to be protected and isolated. Those are called resources and could be memory regions, privileged instructions, devices, or software-defined resources (e.g., pointers, sockets, threads, Inter Process Communication (IPC) buffers, etc.). A protection domain should have a *privilege* in order to be able to access a specific resource. Further, different protection domains may hold different privileges for the same resource. For example, one protection domain could have read and write access to a memory region while another domain could have read access only, for the same region. This could be seen as a finer-grained access control that helps to further enforce the principle of least privilege to overcome the ambient authority problem.

Protection Domain Switch: A protection domain switch is triggered when transitioning from one protection domain with specific privileges to another with different privileges. In task-based systems, this could be task switches or traps from the user to the kernel, while in linkage-based compartmentalisation, this happens between linkage units boundaries. Protection domains are likely mutually distrusting.

Privilege-ring separation: relies on the underlying processor to provide separate privilege rings. This assumes a hierarchy of (dis)trust such as firmware, hypervisor, OS, and applications, all running in separate privilege rings. Less trusted software needs to perform system calls to request a service from a more trusted piece of software. This is considered a coarse-grained protection mechanism and is often associated with privilege-escalation attacks and ambient authority problems. Further, embedded systems do not often map to this trust hierarchy, but they have multiple safety-critical compartments from different stakeholders, each needing its own confined privileges. Thus, privilege-ring separation is often not fine-grained and not scalable or feasible for embedded systems.

Compatibility: For a security architecture to be compatible, existing software already written in a particular programming language should be seemingly straightforward to compartmentalise. There have been a handful of proposed security architectures for embedded systems [26–28, 35, 36], but very few got traction, especially in industry, due to the burden of having to refactor the software around new concepts and designs. A compartmentalisation framework would be source-code compatible if, for example, it does not require changes to the source code or the implementation and design of an existing project. Compatibility is an important requirement to reduce the amount of development and maintenance efforts in order to secure an existing, possibly large, mainstream software project. This enhances the time-to-market as a further non-technical requirement when adopting a new security solution.

Scalability: A secure system should scale with an increasing number of assets and resources that need to be isolated from each other. A system is scalable if it does not have restrictions on the number of compartments or resources. Further, the increasing number of compartments should not noticeably affect the performance or the security of the overall system.

Granularity: The smallest unit or resource that can be isolated determines the security granularity. For example, an MMU protects pages, often at 4 KiB sizes, while MPUs can protect memory regions starting from 32 bytes in size. Also, the ability for protection domains to hold different privileges for the same resource contributes to how fine-grained the security mechanism is.

Dynamic Loading and Security Policies: Some architectures allow compartments to be dynamically loaded and added at runtime. For example, in task-based compartmentalisation, a task can be created, granted some privileges and deleted at runtime. Similarly, dynamic linkage-based compartmentalisation could load, link, and attach privileges to libraries at runtime with the help of a dynamic compartment loader. This could prove useful for systems that need to be more flexible and dynamic, but this could be more complex compared to static solutions where everything is defined at design time, including the compartments and the security policies, which can not be changed at runtime.

2.5.2 Hardware Support for Compartmentalisation

There are two main state-of-the-art hardware technologies to offer protection and compartmentalisation: MMU and MPU.

2.5.2.1 MMU-based Compartmentalisation

MMU-based compartmentalisation is a mechanism to isolate compartments using the MMU and possibly provide shared memory between them. Task-based compartmentalisation (like in Linux and FreeBSD) uses the MMU to enforce isolation between processes. Server-based OSes like UNIX, Windows, and microkernels use the MMU and a well-defined process model to implement software compartmentalisation. Such model (inspired by Multics [30, 69, 70], time-sharing, multi-user systems) was designed around untrusted *users* and a trusted *supervisor* or a kernel. Hardware and software work together to isolate the kernel from the users and users from each other. This could be seen in the current processors having a few privilege rings to protect the kernel from the user and an MMU-based process model to isolate different user's processes from each other. MMU-based systems have advanced since then to meet more resource-hungry user requirements such as virtualisation, paging, large address spaces, and support for many IO devices. Further server-level application security requirements pushed for more fine-grained compartmentalisation and sandboxing [62, 66]. For example, to sandbox JavaScript engines in browsers and to isolate tabs from each other [71] or to compartmentalise OpenSSH to reduce privilege escalations [64].

2.5.2.2 MPU-based Compartmentalisation

MPUs are the state-of-the-art hardware protection mechanism in embedded systems. MPUs configure a fixed number of memory regions (usually up to 16) with base, bounds and permissions. The protection model requires two privilege rings to separate memory regions belonging to an application from the trusted supervisor or kernel.

MPUs suffer from multiple limitations. They are designed with the assumption that the underlying software is either privileged or not. That is, it protects trusted supervisor resources from untrusted applications. While it is still possible to use the MPU to isolate applications from each other, this incurs considerable performance overheads, especially when the number of memory regions and resources exceeds the number of programmable MPU memory regions. The MPU has to be dynamically reconfigured on every protection domain switch, which is a costly operation. Further, emulation is required when there are resources per application more than what the MPU implementation provides. This suffers from performance and determinism drawbacks.

On the software side, using the MPUs to provide protection requires considerable rewriting of an existing OS to isolate its memory from the user applications. This could be a disruptive change to the implementation and design of the OS. Further, some OSes (e.g., TockOS and FreeRTOS) have to provide a new API to use the MPU to allow the user to define a limited number of protected memory regions for an application. This requires the application to be redesigned and reimplemented to use this new API, and thus is not backwards-compatible, source-code wise.

With only coarse-grained privilege rings, accesses to other resources are coarse-grained to only allow the user or the kernel. For example, user compartments can not access system registers. This puts a restriction on compartmentalisation as MPU reconfigurations can not be performed at an unprivileged processor ring, thus having to trap to the kernel, incurring a performance hit. Further, interrupt handlers (or at least the low-level parts of them) have to be executed in the kernel-mode (executing in a more privileged processor ring) then switching back to the user—another performance and design complication that prevents compartmentalising low-level interrupt handlers or make them completely work in a user compartment. Finally, MPUs try to increase the number of regions as a halfway solution to increase the number of compartments. This comes at the cost of more hardware area and performance and power hits when checking memory accesses against an access control table, including the configured MPU regions.

2.5.3 Software Compartmentalisation Techniques

There are multiple mechanisms to enforce software compartmentalisation targetting all or specific software stack subsystems. We discuss them in the following section while defining some terminology that we use across the remainder of this dissertation.

Task-based compartmentalisation relies on a well-defined OS abstraction such as processes, threads, or tasks to define what a protection domain or a compartment is. Communication and sharing between compartments are also defined by the underlying OS and often referred to as IPC.

Linkage-based compartmentalisation relies on linkage units such as object modules or libraries that can be linked (statically or dynamically) to represent protection domains. Communication and sharing can be specified externally by the user at compile or load time or implicitly using the language or linkage models (i.e., symbols visibility, pre-emption, scope, external calls, etc.). It does not rely on some OS abstractions like in task-based compartmentalisation.

Manual compartmentalisation is the process of manually splitting up a monolithic system into smaller compartments. It is often the responsibility of the developers to analyse the entire existing mainstream software, draw logical boundaries and communication bridges between them, then manually refactor the source-code to map those compartments into tasks while allowing communications between them by using the underlying IPC mechanisms. Task-based, for example, is manual as it requires developers to redesign and reimplement mainstream software to use tasks. This does not often scale with large complex mainstream systems.

Automatic compartmentalisation takes a security policy and defined compartments as inputs and automatically enforces compartmentalisation between them. Mostly, no source-code changes are required in this case. Linkage-based compartmentalisation, for example, is automatic in that sense.

2.5.4 Software Subsystems to Compartmentalise

Application compartmentalisation is the process of compartmentalising an application or user level software system. Further OS-related components like device drivers, TCP/IP stack, file systems, memory management, etc., are not involved. For example, an FTP server or a browser can be considered applications that can be further compartmentalised (e.g., by making directories or browser tabs as separate smaller compartments.)

System compartmentalisation spans all the aspects of the operating system and applications, including device drivers, TCP/IP stack, file systems, etc., part of the compartmentalisation process. For example, the process of splitting up a large FreeRTOS monolithic system into smaller library compartments such as the kernel, device drivers, TCP/IP stack, cryptography, communication, OTA, and application is considered system compartmentalisation.

2.5.5 Software Compartmentalisation in Embedded Systems

Unfortunately, embedded system designs and requirements are substantially diverse and different from server-based compartmentalisation. There are no well-defined notions of (multi) users, processes, and a trusted kernel. Further, the main requirements are to have a simple, small, power-efficient, available, real-time system. Historically, applications and OSes are often working together without a clear boundary between what is trusted and what is not, and isolation was not a concern.

Embedded systems got more sophisticated and feature-rich in the era of IoT, with a constant pressing need to connect every *thing* and make each *thing* smarter while being able to respond to external events in real-time. Thus, the design and implementation of such systems have changed to integrate multiple, potentially third-party, feature libraries (e.g., communication, cryptography, user UI, new device drivers, etc.) besides applications and the operating system. Still, most such evolving systems chose not to consider security in order to meet the other historic requirements such as compatibility, determinism, scalability, quick response time, and time-to-market.

With the emergence of software attacks on embedded systems, system designers and researchers tried to use off-the-shelf MPUs and MMUs to get some form of security. However, both approaches do not meet all the embedded system requirements at the same time and hence have not been successful in practice. MMU-based embedded systems had to suffer from the complexity of managing the MMU and page-tables while not using most of its features such as virtualisation, paging, caching configurations, etc. Further, MMUs, caches and Translation Lookaside Buffer (TLB)s introduce indeterminism, having to make space for new code and data by kicking out other entries. A TLB miss could incur significant cost having to walk in-memory page tables. This violates the real-time aspect and determinism requirements. Moreover, conventional MMUs are relatively coarse-grained and provide protection at 4 KiB page granularity, the smallest. This requires multi-level page tables to be allocated in memory and trigger internal fragmentation for software that does not require this much space. In a low-resource memory-constrained system, this will not work. Finally, MMUs and caches take considerable silicon area and consume power, which might not make it a good fit for devices that operate on batteries. The second approach is to use the MPU (see Section 2.5.2.2) hardware feature to enforce software compartmentalisation, but this approach comes with its own limitations due to the inherent MPU design flaws, being a stripped out version of an MMU that was originally aimed for server-based OSes.

2.6 Software Fault Handling

Fault handling is one of the software fault tolerance stages after fault detection. With regards to embedded systems, it is important to recover from a fault in a compartment without affecting the remaining system in order to maintain the availability and reliability of the system. In UNIX-based systems, the focus is on confidentiality and integrity, so it is sufficient to send a signal or kill a compartment on faults. However, in safety-critical embedded systems, availability could be a major requirement. That is, it is not enough to catch the security faults between isolated compartments, but also to be able to recover from or restart the faulting compartment.

Task-based compartmentalisation like in UNIX and microkernels often forwards a fault signal to a task-specific fault handler. This allows for more flexibility, and custom fault handlers depending on the process or task itself. Such technique is custom to the underlying OS design and the faulting compartment; thus, it requires some development efforts and is not generic. Some OSes try to come up with more generic fault handling techniques that are not process-specific. For example, MINIX 3 [72] introduces two user-level servers called *reincarnation* and *data store* to help with general fault handling. The idea is to keep monitoring processes and device drivers to make sure they are alive and functioning properly, and if not, try to reset their state to a previously working state (saved in the data store). This is obviously very customised to MINIX 3 design and requires both memory space and relatively complex logic in the servers, which might make it a not-very-ideal solution for embedded systems.

Microreboots [73] is a cheaper and faster technique to recover from software faults compared to full reboots. Thought to be OS-agnostic, it enhances the availability of the system by not affecting other potentially critical compartments while restarting the faulting compartment. Microreboots help in recovering from unknown faults without requiring a custom fault handler implementation. This is important when it comes to compatibility, but not necessarily performance, compared to other techniques. Microreboots require a well-defined compartmentalised system—which is a challenge in embedded systems. Further, compartments should be relatively small and simple to reduce recovery and start-up time.

Some programming languages and run-times like in C++, Java, and ADA add fault handling capability in a *try-catch* manner. If a fault or exception happens in a *try* block of code, the program flow will be changed to jump to the *catch* block; both defined by the developer. By design, this prompts programmers to write their code with the possibility that faults will occur. However, this is a tailored technique to the programming language itself and is not generic.

Further, it relies on the developer's knowledge and judgment to implement program logic while thinking of potential vulnerabilities that could trigger an exception. This might not work well defending against unknown zero-day vulnerabilities.

2.7 Dynamic Linking and Loading

Linking [74] is the process of combining multiple linkage modules such as object files (e.g., a.o) or libraries (e.g., b.a or c.so) into a single-address-space program image. It is one step after compilation where source code is compiled into machine code and stored into linkage modules. During this process, the linker fixes up external references from one module to another with the correct corresponding addresses in memory.

There are two types of linking; static and dynamic. Static linking all happens during build time by a static linker which is part of a compiler toolchain. For example, ld (The GNU Linker) [75] and lld (The Low Level Virtual Machine (LLVM) Linker) [76] are both static linkers. The output of those is either static/dynamic libraries or an executable format (e.g., ELF [77]). On the other hand, dynamic linking [77] occurs during runtime after the program has been loaded and started. Dynamic linking could be part of a Dynamic Loading [78] process, where linkage modules could lie in a filesystem, ROM, or a network. At runtime, the dynamic loader is responsible for finding linkage modules, allocating memory for them, and performing the dynamic linking stage (at least once).

Dynamic linking and loading have different semantics and goals between general-purpose OSes and embedded systems. For example, in UNIX, dynamic linking and loading are mainly used to be able to share libraries between different users and programs without having to load them for each program. This reduces the memory footprints significantly. Another benefit of dynamic loading is to be able to perform software updates by reloading libraries without having to relink all programs that rely on them, which could prove a scalability issue. This process is relatively complex and costly (in performance and memory size consumption) in UNIX OSes and requires MMU for virtualisation in order to make things transparent for an application programmer. For that reason, dynamic loading and linking are not common in embedded systems. Further, embedded systems do not have a requirement to share or clone libraries; all libraries are executing in a single-address-space environment without the need for virtualisation or unloading libraries. Embedded systems tend to be less dynamic once all software libraries are loaded and linked. However, with the emergence of IoT, the need for connectivity, and the introduction of software attacks, embedded systems required a way to be able to perform software updates to fix bugs or add more features. Some technologies such as OTA [79] updates perform complete updates of the entire software image and require a full reset. This could technically work fine for non-safety-critical embedded systems that do not have availability as one of the main requirements. Examples of OTA are mobile OS updates, laptop firmware updates, or security camera firmware updates. On the other hand, there are safety-critical embedded systems (e.g., in space) that cannot incur a full reset while updating a library to fix critical vulnerabilities or adding features. Availability and reliability are one of the main requirements for such systems, and thus they need dynamic loading and linking to be able to partially and selectively update a library.

Our approach across this thesis does rely on dynamic linking and loading to enforce compartmentalisation between linkage modules. We call this linkage-based compartmentalisation. It is convenient as it already provides linkage units that we treat as compartments, while bridges and communications between them are mapped into API references (e.g., function calls). The security policy can further be provided in a default or custom way and could be updated at runtime. While uncommon in embedded systems, linkage units such as libraries can be unloaded in cases like killing a malicious compartment. In the following section, we describe *libdl* as a dynamic loader and linker for embedded systems. We build on libdl, part of the evaluation and prototyping to provide a secure compartments loader.

2.8 LIBDL: Dynamic Linking and Loading Library for Embedded Systems

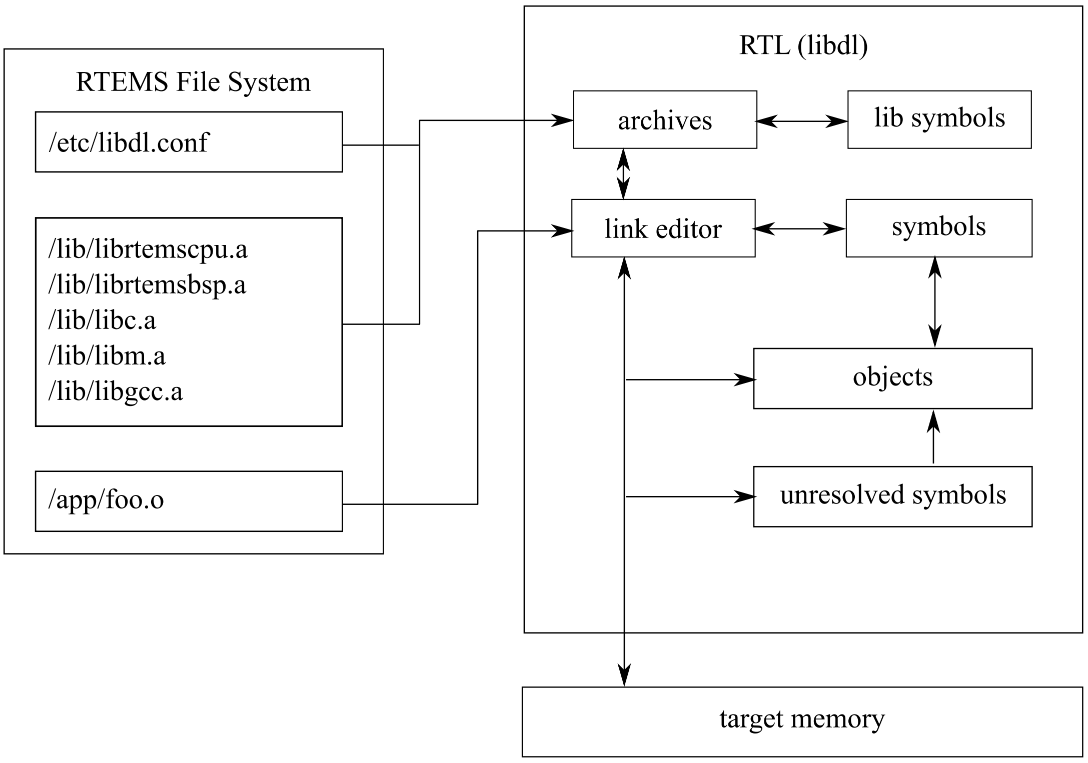


Figure 2.3: LIBDL: Dynamic Loading and Linking Process in RTEMS. The picture is copied with permission from the libdl author and maintainer, Chris Johns. The picture is from [80].

`libdl` [80] is a software component initially developed by the RTEMS project [46] to dynamically load and link ELF modules against a single-address-space, MMU-less, statically-linked ELF image. After the loading/linking process is completed, the resulting performance should be the same as if the ELF image was generated at build-time with static linkers. `libdl` is useful in patching, fixing bugs, updating libraries, and adding more features (e.g., applications, drivers, and libraries) without having to reset the entire system, affecting its availability, as in conventional software updates. The process of linking and loading is described in Figure 2.3. libdl, however, does not provide protection between loaded modules. The main reasons are the inconvenience of using an MMU in embedded systems (e.g., non-determinism, fragmentation, and complexity) and the inefficiency of MPUs as being non-scalable and not providing fine-grained security. In Section 4.3, we describe how we extend libdl as a secure loader to enforce the compartmentalisation part of the CheriFreeRTOS prototype that implements the CompartOS model.

2.9 Capability-based Systems

Since the early days of computer systems, the protection and security of programs have remained a matter of concern. In modern computing systems, different users and programs share the same hardware resources while communicating with each other. This setting requires multiple forms of sharing, such as sharing the processor time, memory, and devices. Resource sharing poses a threat model that might affect the integrity, confidentiality, or availability of the overall system. Protection and security need to be established in order to ensure the correct execution of programs while also retaining the integrity and confidentiality of the system.

Formally, the protection of a computer system is first defined by Lampson in 1974 as "all the mechanisms which control the access of a program to other things in the system" [81]. Following this approach towards security, several access control policies have been introduced to address the protection challenges on shared resources. Lampson has described the access control matrix, which is still used in UNIX-based Access Control List (ACL) and capability-based systems (C-list). In ACL implementations, the system has a central table in which each object has a list of domains (or users) with access permissions attached to them. The table has slots for all possible mappings between domains and resources, which makes it relatively big; the more protection domains and resources in the system, the bigger the table is. Different from ACL, another approach towards security is based on capabilities. In a capability-based system, each protection domain has a capability list of objects it has access to.

A capability, in general, is an unforgeable token to an object in the system that authorises its holder to access that object with a set of permissions embedded in the capability itself. Thus, a capability serves as both an identification (unlike ACL systems in which identities are separate from resources in the access lists) and an authorisation mechanism within a protected system. The capability system itself is only responsible for the integrity of capabilities (i.e., they cannot be forged) and serving requests to create, copy, and revoke capabilities.

The properties of a capability give it some advantages over the ACL systems. Capability-based systems address some of the issues like the size of the table and the requirement to have a list of resources for each domain, which ACL systems fail to achieve. Furthermore, capabilities inherently adopt the notion of *intentionality*; an entity (such as a process) that has a capability to an object is only allowed to access this object with limited access permissions to do its job. This solves challenging issues in ACL systems such as the *confinement* [68] and *confused deputy* [82] problems.

To give a simple example, suppose a user that has ownership access to a confidential file executing a third-party program that might corrupt that file, intentionally or not. The program is not confined (that is, it can access other resources in the system) because it is executing on behalf of the user who is allowed to access other files. Therefore, it is a legitimate action from the ACL system perspective, while in reality, it is not. That is, it is not the intent or desire of the user for the application to be able to access (and potentially corrupt) the file. What makes more sense is to give the program only the set of capabilities to resources it needs and restrict the operations that it can do on them, sufficient enough to finish its job. That is also defined in the literature as *the notion of least privilege*.

Capability-based systems give some practical solutions to security and protection issues such as:

- The confused deputy.
- The confinement problem.
- Scalable access control management.
- Fine-grained access control.

There have been significant efforts to build capability-based systems in software, hardware, programming languages, or a combination of them. This section discusses the most influential capability systems that have been built, most of which are introduced in Levy's capability-based systems book [83].

2.9.1 Hardware Capability Systems

Dennis and Van Horn first defined the word "capability" in their paper that got published in 1966 [84]. The paper introduced new concepts back then, such as a supervisor, capability lists, processes and domains, all of which are still used until now, more than fifty years later. The authors proposed a hypothetical capability-based computer system called "the supervisor" and explained how it could be used for multiprogramming computations. However, it did not include any actual implementation or evaluation of the proposed system. In this system, a protection domain consists of a list of capabilities, which gives access to the resources the domains need to finish a computation. A capability merely names an object with permissions attached to it. In order to make the system generic and abstract, the authors suggest not to restrict object names to memory addresses. The paper then goes on with defining what an object is and which operations can be invoked on capabilities. Only three types of capabilities are specified in the system: directories (for naming permanent objects), segments (for memory protection) and entries (for executing or transferring to another protected procedure).

It was not until the MIT PDP-1 [85] system that an actual implementation adopted Van Horn's capability system. PDP-1 had the conventional C-list stored in fixed locations at the start of each process' address space. Capabilities refer to high-level resources such as terminals and tapes, while there is room for adding new object types. The timesharing system was in use by students until the mid-seventies.

In Levy's book about capability-based systems [83], he mentioned another uncompleted hardware project called The Chicago Magic Number Machine. The project started in 1967 with the goal of producing a general-purpose capability-based system. In Chicago's machine, memory protection was only enforced by capabilities (i.e., no MMU). Chicago's machine is very similar to the modern CHERI architecture, which will be discussed later. It has separate registers for capabilities other than general-purpose data registers, and they can also be stored in special capability segments in memory. Each capability has base-and-bounds, type, and access attributes. Unfortunately, it has never been deployed.

In the next decade, the Cambridge CAP Computer [86] was developed at the University of Cambridge. The project was led by Roger Needham and Maurice Wilkes. Unlike Chicago's machine, the CAP computer was completed and fully functional with an operating system and toolchain support. The CAP computer had sixty-four entry capabilities; however, those capabilities cannot be programmed by the user. Capabilities can be saved in the capability unit or memory in a segment that only contains capabilities, similar to Chicago's machine. Each process had its own capability segment, which can access other system's objects. The CAP capability format consists of a base, size, and access permissions. Memory access instructions should specify a segment capability. The virtual address in CAP indexes segment capabilities and an offset within it. That is analogous to how MMU's virtual addresses index pages.

The efforts spent in building hardware-based capability computers were ceased with the emergence of CISC-based PC architectures, MMU, UNIX, and microkernels. MMU- and CISC-based processors were very tempting from the performance perspective to build monolithic kernels such as UNIX.

That was fine until attacks on the MMU and CISC computers started to emerge. The reasons have been mainly because of design weaknesses of MMU and their implementations, as well as the increased complexity in CISC systems that introduced bugs. Complexity is the enemy of correctness, and that is why the current research community is focused on RISC and microkernel systems that can easily be reasoned about.

2.9.2 Software Capability Systems and Microkernels

Hydra [87] was the first general-purpose object-based capability system. It was developed at Carnegie Mellon University. The primary motivation for Hydra was to allow operating systems research and extensibility. Some of the design choices like putting drivers in userspace and separating policies from mechanisms in the kernel found their way to microkernel designs [88] and are still being adopted in modern L4 microkernels such as seL4 [89] and Fiasco.OC [90]. Hydra provided a new system abstraction at the time, making everything in a computer system an object.

EROS [91] introduced a new idea of revoking capabilities by versioning objects and capabilities pointing to them. To revoke access to an object, the version of the object would be changed, and consequently, the mismatched versions (during a dereference) will trigger a fault. Capabilities in EROS are represented as nodes. A protection domain consists of a tree of nodes of capabilities. Each node has a fixed size of thirty-two capabilities. Unlike seL4 (described next), EROS tries to map pages on virtual memory faults, and if it fails, it calls a user-level fault handler in a capability. seL4 always redirects faults to the fault handler in userspace.

seL4 is a modern microkernel with a focus on security within embedded systems. The authors of the 2009 seL4 paper claimed it is the first general-purpose operating system to be formally verified [89]. This means that the high-level kernel specification matches the C code (and in later versions, the binary itself). With the assumption that the assembly code and the hardware are correct, seL4 is claimed to be bug-free and would never crash. This comes with a few caveats and assumptions [92]. For instance, seL4 does not currently protect against timing channels, DMA-related attacks, or further exploitable hardware vulnerabilities. It also only proves the integrity and confidentially of the system with those caveats.

Adopting microkernel principles, seL4 embraces simplicity in its design and implementation. This enabled the trusted codebase to be small enough (less than 10000 lines of code) that complete formal verification of the kernel behaviour in every possible path is performed and reasoned about to be bug-free. This effort narrowed the gap of having a trustworthy software system. The downside, still, is that the hardware is assumed to be correct, which is not usually the case. For example, the verified seL4 code is vulnerable to recent covert channel attacks such as Spectre [93]. Furthermore, seL4 relies on conventional MMU to provide isolation, which largely lacks determinism and have coarse-grained memory protection granularity of 4 KiB. That is, seL4 will not be a good fit for embedded systems that require determinism and small size protection units (e.g., 4 bytes MMIO registers).

Capsicum [66] is a software capability API added to UNIX in order to address the compartmentalisation requirements that discretionary [94] and mandatory [95] access control mechanisms cannot handle. Capsicum addresses the first broad-rights issue restricting the rights by wrapping file descriptors in a new capability file descriptors type. It was implemented as an extension to UNIX's file descriptors in the FreeBSD kernel. Unlike capability-based microkernels and pure capability systems, Capsicum capabilities work together with the existing UNIX's mechanism in a hybrid way to provide fine-grained access control and application-level sandboxing solutions. Similar to capabilities, file descriptors index unforgeable tokens of authority to objects and can be delegated over an IPC channel. However, file descriptors have broad rights. That is, their access permissions are coarse-grained. Moreover, file descriptors delegation is still controlled by UNIX-based IPC mechanisms such as sockets, which are still under ACL enforcement [96], unlike pure capability systems.

2.9.3 Language-based Capability Systems

Another category of capability systems targets only compilers and programming languages.

Joe-E [97] is an object-capability language built on top of Java with the goal to develop secure systems. Being built on Java, Joe-E is a memory-safe language that decreases the memory attack surface that exists when developing C/C++ based systems [51]. Joe-E addresses higher-level security challenges by motivating the programmer to inherently write secure applications in a "security by design" approach. This is achieved by writing code with capability principles in mind. Joe-E takes advantage of the object-oriented aspect of Java; it deals with object references as capabilities and puts a set of design restrictions in order to control the propagation of references and how they can be delegated. For example, it is prohibited to define native methods. This ensures references are unforgeable. Another decision is to modify Java's APIs to use capabilities in order to prevent the ambient authority problem. Joe-E does not, however, strictly follow pure capability design as it does not have access permissions associated with references.

Similarly, Caja [98], developed by Google, is a compiler tool to sandbox untrusted third-party packages written in HTML, CSS or JavaScript in websites.

Capability-based programming languages target application-level security rather than the whole hardware and software system. CHERI, however, aims to leverage capability-based security to the whole system, including the hardware ISA, operating systems, hypervisors, user applications and programming languages.

2.9.4 Capability-based Systems Limitations

As discussed, capabilities generally improve security. However, moving from legacy, non-capability systems, to using capabilities, comes with costs and overheads. For instance, it takes non-trivial efforts, resources, and time to build sound capability-based core systems such as seL4 and CHERI. Once built, users of such systems have to gain the knowledge and understanding of how capabilities are represented, how to handle them, and how to safely propagate them. Such operations differ from one capability systems to another (i.e., there is not a unified interface for "capabilities"). Later on, legacy software has to be ported with the notion of capabilities. This could be an error-prone process, depending on how sound software developers grasped the concepts and interfaces of the underlying capability-based systems. The porting process could be non-trivial for systems such as seL4 that require manual refactoring of the existing software to use capabilities, especially when written in millions of lines of code. Finally, debugging a capability-based system might not be intuitive. Understanding capabilities errors messages and adding support for debuggers to understand capabilities could be quite challenging.

As we discuss next, CHERI and CompartOS aim to make it easier for system designers and developers to entertain capability-based security in an automatic and compatible manner. This helps in addressing most of the challenges and overheads we mentioned.

2.10 The CHERI Architecture

In this section, we discuss the CHERI architecture [61, 99] and the potential protection models that can be of use in embedded systems. CHERI is a modern capability-based RISC architecture that is being developed by the University of Cambridge and SRI International. The main principles CHERI is designed around are:

- *Principle of least privilege*: which motivates the idea of reducing privilege rights to a piece of software as much as possible.
- *Principle of intentional use*: by naming the privilege a software application uses, rather than giving it full privilege accesses and let it choose what privileges to use implicitly.

CHERI is designed with these principles in mind such that a software application that runs on top of CHERI inherently maintains capability attributes. This helps in reducing the access rights an attacker has and consequently minimises the attack surface.

2.10.1 Overview

In this section, a general overview of the CHERI architecture is discussed, along with some of its protection properties. CHERI is designed to be architectural-neutral, meaning that it can (and has been) easily mapped to a RISC ISA such as MIPS, ARM, or RISC-V. At the architectural level, CHERI provides in-address-space memory capabilities on all pointers. For example, all software pointers in a C-based application are represented as capabilities and protected in hardware. Each pointer has a base-and-bound as well as access permissions, depending on its use. On dereferencing a pointer, CHERI performs hardware checks and traps on bounds and access permission violations. This is an application of the two main principles that help to protect against conventional buffer overflow attacks as well as minimise the effects of software bugs.

The main architectural additions for a specific ISA to support CHERI are capability registers, tagged memory support, and CHERI's instruction set.

2.10.2 CHERI Registers

CHERI has a number of general-purpose capability registers that are either separate from the underlying processor's general-purpose registers (GPRs) or extend them (i.e., unified register file). Recent CHERI implementations such as CHERI-RISC-V (see Section 2.10.5) favour the unified register file approach as it is less costly in hardware and software deployment. The unified register file is usually in a compressed hardware format [100] that doubles the underlying general-purpose registers in order to represent capabilities rather than just integer addresses. For example, 64-bit GPRs become 128-bit capability registers that can hold either legacy integer addresses or CHERI capabilities. Similarly, 32-bit GPRs become 64-bit capabilities. The CHERI capability content is as follows:

- **Base:** is the start address of the memory region a capability covers.
- **Length:** is the size of the memory region.
- **Offset:** is a pointer inside the region, for example, an offset within an array.
- **Permissions:** Contains architectural permissions such as read, write, and execute.
- **User Permissions:** Software permissions that can be defined and used by the user.
- **Type:** a user-defined type that takes effect when a capability is *sealed*.
- **Sealed:** when set, it indicates that a capability is software-defined (i.e., not a memory/-pointer capability).
- **Tag:** marks a capability validity.

Apart from the general-purpose registers, there are also a few special-purpose registers:

- **DDC:** Default Data Capability is implicitly being checked against during every non-CHERI load/store in compatible (or hybrid) mode.
- **PCC:** Program Counter Capability contains the bounds and permissions the underlying running code is allowed to run with.
- **EPCC:** Exception PCC holds the value of the interrupted PCC during traps.

On processor reset, ***DDC*** and ***PCC*** are initialised by the hardware to cover the entire address space with the proper permissions. They are the root of all other capabilities that might get created. Later, a secure bootloader, a hypervisor, or the operating system might choose to decrease permissions and bounds for both registers and manufacture less privileged capabilities out of them to hand over to an application thread, for instance. It is assumed that the first software (or firmware) that is taking control after the processor reset is secure, trusted, and CHERI-aware. We show later in Section 4.3 in our CheriFreeRTOS implementation and evaluation that a secure boot loader takes full control of PCC and DDC and creates restricted capabilities out of them to be handed over to the OS, libraries, and applications.

2.10.3 Tagged Memory Support

Each capability in CHERI is associated with a tag, which indicates whether this capability is valid or not. In-memory capabilities have the tag bit in a separate unaddressable tagged memory. That is, the tagged memory support, part of the memory subsystem, has a bit for each capability-aligned memory word that is being checked during loads and stores. The tag bit is set when a valid capability in a CHERI register is stored in memory. Right after the processor reset, all the tags are cleared. All valid capability operations maintain the bit, including deriving less privileged capabilities (i.e., with fewer permissions) or capabilities to smaller memory regions, which guarantees the provenance of capabilities (i.e., that each one is derived from one of the initial capabilities). The operating system boot code can then derive capabilities from ***DDC*** and ***PCC*** and save them in memory.

On memory loads and stores, the tag bit of the capability used is checked, and an exception is triggered if the tag is unset. This ensures the integrity of CHERI capabilities. If a normal store (i.e., non-CHERI stores) writes to a memory address that happens to have a capability, the tagged memory subsystem clears the corresponding tag bit. For example, this prevents a shell-code injected as data and cast to a pointer function from being executed as there will not be a valid tag for that pointer.

2.10.4 CHERI Instructions

CHERI has an instruction set to retrieve and manipulate capabilities contents. These instructions are used to create, modify, or derive capabilities from existing ones. The *monotonicity* property of CHERI is maintained by ensuring, by design, that any of the capability manipulation instructions never gain more permissions or bounds than the ones they are being derived from. The CHERI load/store instructions are used for both capabilities and normal data. All loads and stores, whether CHERI-aware or not, are checked against explicit or implicit capabilities. All CHERI instructions have to explicitly specify the capabilities part of their operands. Another set of control flow instructions are used with code-based capabilities to jump, branch, and return between different subroutines.

To allow for software-defined capabilities and object-model invocations, CHERI provides *cseal* and *cunseal* instructions. Sealing means that a capability is not interpreted as a pointer. It is up to the application to define types and naming schemes of its sealed capabilities. A sealed capability cannot be dereferenced until unsealed. Sealed capabilities can be used to create user-defined types (i.e., capabilities to complex objects instead of memory objects) or to pass capabilities through entities that should not have the right to dereference the capability (e.g., intermediate layers in a software stack).

2.10.5 CHERI Hardware Implementations

The CHERI ISA has been under research for over ten years at the time of writing this dissertation. It has been implemented in multiple software simulators, softcores that run on Field Programmable Gate Arrays (FPGA)s, and most recently, an ASIC processor. Being a research architecture, for the most part, CHERI has been mainly implemented on FPGAs as they provide rapid prototyping (compared to ASIC processors) that allows us to change hardware configurations (e.g., cache size, pipeline stages, frequency), add features and have cycle-accurate performance analysis. That said, performance evaluation on FPGAs is approximate to ASIC, but not the same. FPGAs are generally slower (frequency-wise) than ASIC processor, and they use different technologies to represent logic gates and memories. Benchmarking software on FPGAS (especially using cycles) is still a good indicator of performance if run on ASIC processors with the same processor configurations (e.g., pipeline stages, cache sizes, and DRAM) and should not differ significantly.

2.10.5.1 CHERI-MIPS

The CHERI ISA was first adopted in CHERI-MIPS [101] and was prototyped on the Bluespec Extensible RISC Implementation (BERI) softcore processor [102], which implements a MIPS ISA in Bluespec HDL [103]. CHERI-MIPS is a 64-bit CHERI processor that was targetting general-purpose UNIX systems with MMUs. For example, the main research OS back then was CheriBSD [104] (a fork of FreeBSD). CheriBSD evaluates the use of CHERI to provide pointer safety while it complements the MMU-based protection model. All of the previous published CHERI research on CheriBSD, CheriRTOS, CheriOS, and further CHERI software compartmentalisation has been evaluated on CHERI-MIPS.

2.10.5.2 CHERI-RISC-V

Over the past few years, the implementation of CHERI has shifted focus from MIPS to RISC-V [105]. RISC-V has been a relatively new promising ISA (that began in 2010) for both research and industry, and it quickly got traction in the hardware and software communities for being flexible, modern, and extensible. This allowed multiple open-source efforts to implement RISC-V softcores such as the RocketChip [106] and the Bluespec chain of RISC-V processors [107]. CHERI has been building on the Bluespec RISC-V softcores as its main research processor prototype to implement the CHERI-RISC-V ISA [108]. There have been three main CHERI-RISC-V softcore implementations: CHERI-Piccolo, CHERI-Flute, and CHERI-Toooba running on FPGAs. CHERI-Piccolo is representative of low-end MCU and M-Class Arm processors. CHERI-Flute is similar to low-end A-class processors, and finally, CHERI-Toooba is the most advanced one being an out-of-order, multicore, superscalar processor. As we discuss later in the evaluation chapter (see Chapter 5), we have chosen a 32-bit CHERI-Flute softcore to perform our embedded systems and CompartOS evaluations on as it falls within the scope that EOSes we are targetting run on.

2.10.5.3 Arm Morello

There have been no CHERI-based ASIC processors that we can use for the purpose of evaluation during this period of research. That said, there is a new CHERI-based Arm processor (called Morello) that was released in early 2022 [109]. However, Morello is mainly targetting general-purpose operating systems such as CheriBSD/FreeBSD, Linux, and Android. Thus it is not suitable for the scope of our embedded system. Morello is based on Neoverse N1 CPU, which is a high-end microarchitecture used in servers. We hope the efforts in this thesis and particularly the realism of evaluation will encourage industry to consider CHERI-based processors for embedded systems.

2.10.6 CHERI Capabilities Compression Implications

The CHERI compression algorithm [100] adds some alignment requirements on memory allocations. For instance, OS memory allocators need to enforce base and top address alignment requirements to be able to represent returned *malloc*ed regions as capabilities. The current CHERI-RISC-V implementation requires no alignment constraints for allocation less than 4 KiB on 128-bit capabilities. For allocations bigger than 4 KiB, the top and base fields of capabilities have to be aligned to 2^{E+3} where E is determined from the requested length. CHERI provides two helper instructions to efficiently and quickly calculate (e.g., by OS allocators) both the base and top addresses that meet the alignment requirements.

Due to the decreased number of bits in 64-bit capabilities, the alignment requirements are stricter. For instance, in 32-bit RISC-V systems with 64-bit CHERI capabilities (the CHERI-RISC-V system we use for evaluation in this thesis), there are no alignment requirements for allocations less than 256 bytes. Bigger allocations need to align the top and base addresses. Depending on the embedded software workloads and applications, over allocating the sizes of the requested memory regions might be required, and this may introduce internal fragmentation and memory consumption inefficiency, especially for large allocations.

2.11 Software Protection Models

On the software level, CHERI gives the option to isolate software components within an object-based environment. Besides memory capabilities, CHERI allows the software to define its own type, bounds and access permissions. This helps to compartmentalise software with strong isolation enforcement.

2.11.1 Pointer Safety

Pointer safety is the primary application of CHERI in C/C++ languages. This mostly relies on the compiler toolchain to map C/C++ pointers into CHERI's memory capabilities. In the current implementation of CHERI, the toolchain is LLVM/Clang. There are two main modes a CHERI-aware C/C++ code can be compiled in: compatible and pure capability modes.

The *compatible mode* (also known as the hybrid mode) gives the programmer the ability to specify which pointers are represented in CHERI capabilities and which are the conventional integer pointers represented in the underlying ISA registers (e.g., MIPS). An existing C/C++ project can be compiled in CHERI compatible mode, unmodified. CHERI's *DDC* and *PCC* registers are set on startup to cover the entire address space with sufficient permissions. This should be transparent to the programmer. An application that aims to enforce protection on chosen pointers can explicitly tag a pointer with the compiler's *__capability* keyword. Any operation on this pointer is protected by the CHERI architecture. For instance, a C array can be represented as a capability with the base assigned to the start address of the array, and the length equals the size of the array. Permissions can only be read or write.

The second mode is the *pure-capability mode* (PURECAP). In this mode, every pointer is interpreted by LLVM/Clang as a CHERI capability with base, bounds and permissions, pushing pointer safety enforcement to the limits to get stronger pointer isolation guarantees. Pointers include conventional data pointers, arrays, functions, stack and heap allocations, and return addresses. Violations trap to the executive, which decides how to handle the violation.

2.11.2 Software Compartmentalisation

CHERI can also provide the basis for efficient, arbitrarily fine-grained software compartmentalisation. This is an emerging research area and the primary focus of this thesis. Apart from pointer safety discussed earlier, CHERI provides the flexibility for software developers to define their own notion and interpretation of a capability. That is, permissions, naming, and the type of capabilities are defined by developers while still benefiting from CHERI's provenance, integrity and monotonicity properties. For example, in pointer safety, the naming of a resource is basically an address, while in software compartmentalisation, a developer can define the naming scheme of a resource. This can be an actual string name, interrupt number, or any identification method a project may deploy for resources. In Chapter 6, we further discuss CHERI-based efforts as related work that apply different software compartmentalisation models such as CheriABI [33] and CheriRTOS [35].

2.12 Summary

This chapter gave a background on embedded systems and security in general. We discussed the scope of embedded systems we are targetting for the purpose of this thesis, then the state-of-the-art protection mechanisms that have been used to secure such systems. We then defined what software compartmentalisation is, why it is useful, and how it could be implemented using existing protection units. At the end of the chapter, a review and history of capability-based systems were given before giving an overview on CHERI and its protection models.

CHAPTER 3

COMPARTOS MODEL

This chapter describes the CompartOS model and motivates its potential in securing high-end, complex, and mainstream embedded systems. The main hypothesis we are evaluating is that:

CHERI and CompartOS can achieve greater compatibility, security, availability, and scalability than MPU-based and task-based protection for embedded OS compartmentalisation while they only add lightweight performance and development overheads when applied to secure completely unprotected mainstream embedded systems.

The chapter starts with listing the main requirements and goals for CompartOS in Section 3.1 and, when applicable, briefly mention how the model helps meet such goals. In Section 3.2, we describe what the CompartOS model is and the design choices we have made in order to meet the goals. Section 3.3 discusses the types of faults that CompartOS considers and a few fault handling mechanisms that could be integrated into the model. Chapter 4 describes how we implement the CompartOS model in our prototyped software artefacts, and Chapter 5 evaluates this prototype of our model and compares it against state-of-the-art MPU-based and task-based protection models across various microbenchmark, TCP/IP use cases, and real-world, safety-critical, deployed systems.

3.1 Requirements and Goals

This section discusses the requirements and goals that CompartOS is designed to meet and how they are useful, and how they compare to other state-of-the-art embedded security architectures. It also briefly describes how CompartOS meets those requirements, but the thorough evaluations are further discussed in later sections and chapters.

Capability-Based Security: Due to their OS design and limited hardware, embedded systems generally lack an access control model such as ACL and filesystems in UNIX-based OSes. This means that nothing stops an attacker on an embedded system from accessing the entire memory space and all system resources. Capability-based access control provides a scalable, lightweight mechanism that can be implemented with little perturbation to existing software. CompartOS aims to leverage capability-based security [83, 91, 96] everywhere: not only protecting kernel objects (as in capability-based microkernels [89]) but also providing hardware memory protection and programming-language pointer safety. That is, CompartOS extends security enforcement into the compartment itself, rather than just providing isolation guarantees between compartments.

Unlike the current capability-based microkernels, CompartOS is designed around CHERI hardware capabilities as the sole mechanism for protection. That is, CHERI provides a replacement to MPU/Physical Memory Protection (PMP), MMU, and privilege rings protection. Not only that CHERI offers complete pointer safety for CompartOS, but it also allows for further fine-grained object-based software compartmentalisation. Capabilities in CompartOS describe both linkage-level resources and more abstract software-level resources (kernel objects). This ultimately provides programming language memory-safety (within isolated compartments) while maintaining the inherent security, isolation, and scalability virtues of capability-based compartmentalised systems.

Scalable Automatic Compartmentalisation and Security: Composing multiple components, libraries, or applications is increasingly common in embedded systems, with third-party libraries and standards becoming more feature-rich every day. Therefore, increasing the number of resources and features (represented as compartments) should be scalable, without significantly affecting performance and security guarantees or requiring major engineering and maintenance overhead. CompartOS' linkage-based compartmentalisation model is inherently scalable, generally requiring minimum to no additional engineering work to add a component.

Inter-compartment Security Policies: Embedded systems need to define a way to restrict interactions and sharing between compartments. MPU-based, task-based IPC could work fine but could also be costly compared to direct function calls, especially if the mainstream software is not designed around tasks and IPCs. Further, the privilege-ring separation into user and kernel modes may be too coarse-grained for compartments with different criticalities and varied privileges (i.e., beyond privileged or unprivileged tasks, or secure and unsecure modes). For example, there should be a way to allow one client compartment to communicate with a server compartment while preventing all other compartments from communicating with that server compartment.

CompartOS specifies security policies for inter-compartment interactions at build time and enforces them at load-time, and relies on CHERI to inductively extend the security of these initial conditions. It is fine-grained enough to handle C-based resources such as function and object filtering (by having block and allow lists of symbol names), which effectively enforces API boundaries as in syscall (with privilege-ring-based protection) and function-entry based systems. More abstract policies around kernel objects, services, and resources can also be defined. For example, in CompartOS, a user can prevent IPCs from one compartment to another by rejecting minting capabilities to queues or message buffers (that are used for IPCs). Capability minting is creating a copy of an original capability potentially with fewer privileges. Further examples of defined policies could be to:

- Define C-based object and function call boundaries.
- Restrict kernel services for a compartment.
- Prevent IPCs and communication between compartments.

Source-Code Compatibility: Mainstream, large, complex, and unprotected embedded system software could be written in millions of lines of code. Providing security for such systems at a minimum development and maintenance overhead is one of the main requirements. That is, applications should require few, if any, source-code changes.

We deliberately chose to avoid designing a completely new research OS or rewriting existing non-secure systems in memory-safe languages such as Rust, all of which complicate learning, development, runtime, and maintenance overheads, and can be a deal-breaker for large complex mainstream systems in the industry.

CompartOS targets existing and widely used software written in C—such as FreeRTOS. The new compartment model requires relatively minor changes to the existing OS for kernel developers, thus allowing mainstream systems to be secured without significant refactoring and maintenance overheads. Further, existing embedded software systems can be easily compartmentalised by defining what libraries should be compartmentalised and rebuilding them using a CHERI-aware toolchain under CompartOS. This is far less burdensome than task-based compartmentalisation, which requires source-code modifications and refactoring to redesign the system around tasks.

For an application developer, it is just a matter of rebuilding the application and, optionally, defining the security policy among compartments. That is, source-code compatibility (but not binary compatibility as compartmentalised libraries need to be rebuilt) is achieved, given that it is cleanly written and not violating CHERI C/C++ memory safety [110]. Cleanly written code does not attempt to cause buffer overflows or access data or code beyond its bounds. More specifically, it does not perform obscure pointer arithmetic that might cause the pointers to cause accesses beyond bounds. It does not try to forge pointers either. This makes it compatible with CHERI C/C++.

CompartOS and CHERI require having full access to mainstream source-code to be able to rebuild it to enforce CHERI C/C++ memory safety and CompartOS' software compartmentalisation. This means that legacy libraries or programs in binary format (without source-code) cannot be compartmentalised or sandboxed. Rebuilding an embedded software stack is often fine, given that it is modular. Linkage-based compartmentalisation must maintain source-code compatibility. Furthermore, if no inter-compartment security policies are provided, the system will still be compartmentalised by default via a programming language (e.g., C) and linkage semantics (e.g., global and local C-based symbols). This helps in achieving availability and fault isolation and handling between linkage-based compartments.

Improved Availability: Many attacks target the availability of embedded systems [111–115] rather than confidentiality and integrity. General-purpose OSes (e.g., UNIX) have a lower focus on availability and it is often a secondary design goal. In contrast, breaking availability in safety-critical embedded systems could sometimes cost lives, e.g., if they manage medical databases, information on air traffic, and so on. For example, a null dereference or a buffer overflow in a library should not crash an entire system; it should be caught and handled with the minimal possible effect on system performance. Previous (UNIX-centric) CHERI-based work took the first step of catching accidental or malicious spatial memory faults. Building on CHERI, a primary design goal of CompartOS is to improve system availability and, in particular, for safety-critical components. Such system will also generally recover gracefully from confidentiality and integrity attacks on a compartment, improving availability in an adversarial environment. That said, CompartOS focuses more on recovering from software faults (either memory-safety related or others) rather than preventing them. That is, developers can provide their policy and mechanism that can detect a problem (e.g., by using CHERI), while CompartOS can provide custom or general handling of the affected compartment to maintain system availability.

Real-time and Embedded-Systems Requirements: Most mainstream RTOSes still need some form of determinism as a real-time requirement, and that is why they avoid using an MMU to enforce security and protection.

CompartOS does not introduce non-determinism or add to the complexity of the to-be-compartmentalised real-time embedded system itself. For example, there is no required hardware caching, virtualisation, or (re-)configuration. Furthermore, CompartOS does not add additional background software, such as secure monitors or runtime checks (as in intrusion detection systems), which might affect the performance and timing guarantees of the software. Instead, CompartOS relies on the deterministic and lightweight checks performed in CHERI hardware to detect potential violations of its security policies. In addition, CompartOS' system-level assumptions (e.g., threading support and single-address-space execution) are met by most RTOSes.

Compartmentalised Runtime Software Updates and Restarts: Long-running and safety-critical embedded systems need a way to fix vulnerable compartments or add new features without shutting down the entire system. Although optional, CompartOS has two other major benefits—thanks to its dynamic loading and runtime compartment creation support:

- Adding more features, libraries, and compartments at runtime.
- Improving availability by patching and restarting vulnerable compartments.

Current embedded OSes provide a mechanism for software updates, commonly called OTA updates, that allows the system to do a full image update. This approach, however, requires a full restart that effectively shuts down and reboots the whole system—something that is not desirable for most real-time safety-critical systems having availability as a crucial requirement.

In our design, we make sure we can selectively update and (re-)start a compartment at runtime, in isolation from the remaining system that might be running other safety-critical tasks. In some cases, catching an exploitable vulnerability and exercising fault handling mechanisms (as discussed in this dissertation) might not be sufficient, especially with recurrent faults and attacks. Fixing vulnerabilities at runtime will be required, for example, after zero-day vulnerabilities are disclosed. There are two use cases where this might be useful:

1. Long-running real-time safety-critical software running in the space.
2. Short-running software system that needs urgent fixes, updates, and restarts.

Applicability: A security model for embedded systems should be largely applicable and realistic to deploy in industry in order to get traction. That is, it is not supposed to be specific to a particular "research-only" use case or EOS.

CompartOS aims to be generic enough to be applicable, portable, and adaptable to existing embedded (operating) systems and applications. This is achieved by assuming as little as possible about the existing environment:

- Using CHERI as the only hardware feature required for protection.
- Using CHERI compiler toolchain.
- Single-address-space environment.

Threat Model: Embedded systems should assume there is always a zero-day vulnerability or a malicious compartment as history suggests software always has bugs [116]. This assumption leads to proactive defence, with techniques such as compartmentalisation. That way, software systems could preemptively defend against future unknown vulnerabilities as well.

Our threat model assumes an embedded software system with third-party and unverified libraries and applications executing in the same address space without isolation or memory protection. We refer to libraries and applications as compartments, which can be added at build-time or runtime. The Trusted Computing Base (TCB) is kept as small as possible and consists of the kernel and a runtime compartment secure loader. The toolchain and supply-chain are not trusted, unlike systems written in Rust, for example. Everything else, such as device drivers, OS libraries and services, and applications are untrusted, and they are also considered compartments.

Untrusted compartments could attempt to violate (deliberately or unintentionally) language-based memory-safety guarantees enforced by CHERI, OS-level isolation mechanisms, or externally defined inter-compartment security policies. Like other isolation technologies, we assume the attacker seeks to compromise the system's confidentiality and integrity. Unlike other technologies, we also assume the attacker seeks to disrupt system availability, and we explicitly design and evaluate against this target. We further assume a CHERI-aware attacker that is either resident on the device (e.g., actively malicious and controlling a library) or in direct communication with a vulnerable software component on the device (e.g., the TCP/IP library) but has not compromised the TCB.

OS and hardware level attacks are beyond the scope of our threat model. For instance, attacks that target scheduling and timings (e.g., hogging the CPU) or side channel attacks are not considered for the purpose of this thesis.

Ease of Debugging: Embedded systems should still be easy to debug after adopting a new security solution. This reduces maintenance and time-to-market overheads. One of the main criticisms of capability-based microkernels and other componentised research OSes is the lack or difficulty of debugging support. In CompartOS, ease of debugging is a main design goal. Debugging compartmentalised systems in CompartOS is as easy as debugging baremetal statically linked programs, thanks to the single-address-space design assumption, and to shared-library and dynamic-loading support in GDB. Furthermore, a complete stack trace is optionally available on per-task faults showing details about the fault, compartment switches, and function calls.

3.2 Compartmentalisation Model

This section describes CompartOS as a generic compartmentalisation model and its design choices. The model is not specific to a particular implementation but can be applied to mainstream embedded (operating) systems, boot loaders, and programming languages. An overview of a CompartOS model system is shown in Figures 3.1 and 3.2. Both figures are further described in the design choices discussed next.

CompartOS is a programming language- and linkage-based compartmentalisation model.

CompartOS relies on the underlying linkage model and format (e.g., ELF, Mach-O, or PE) to define what a compartment is at compile time. The security policy between compartments

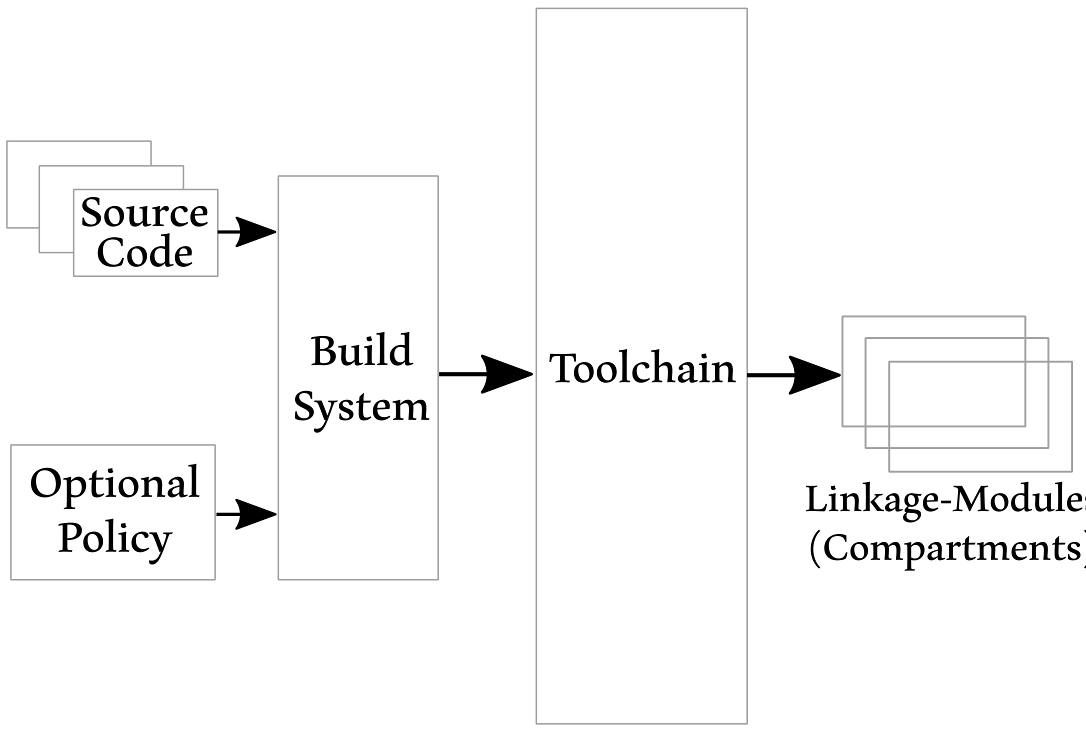


Figure 3.1: CompartOS build-time workflow. Developers feed the (mainstream) source-code to the build system as well as an optional security policy that splits up the source-code into logical compartments with well-defined APIs (e.g., libraries), along with the restricted communications between them. The output is linkage modules that are fed to the dynamic secure loader at runtime.

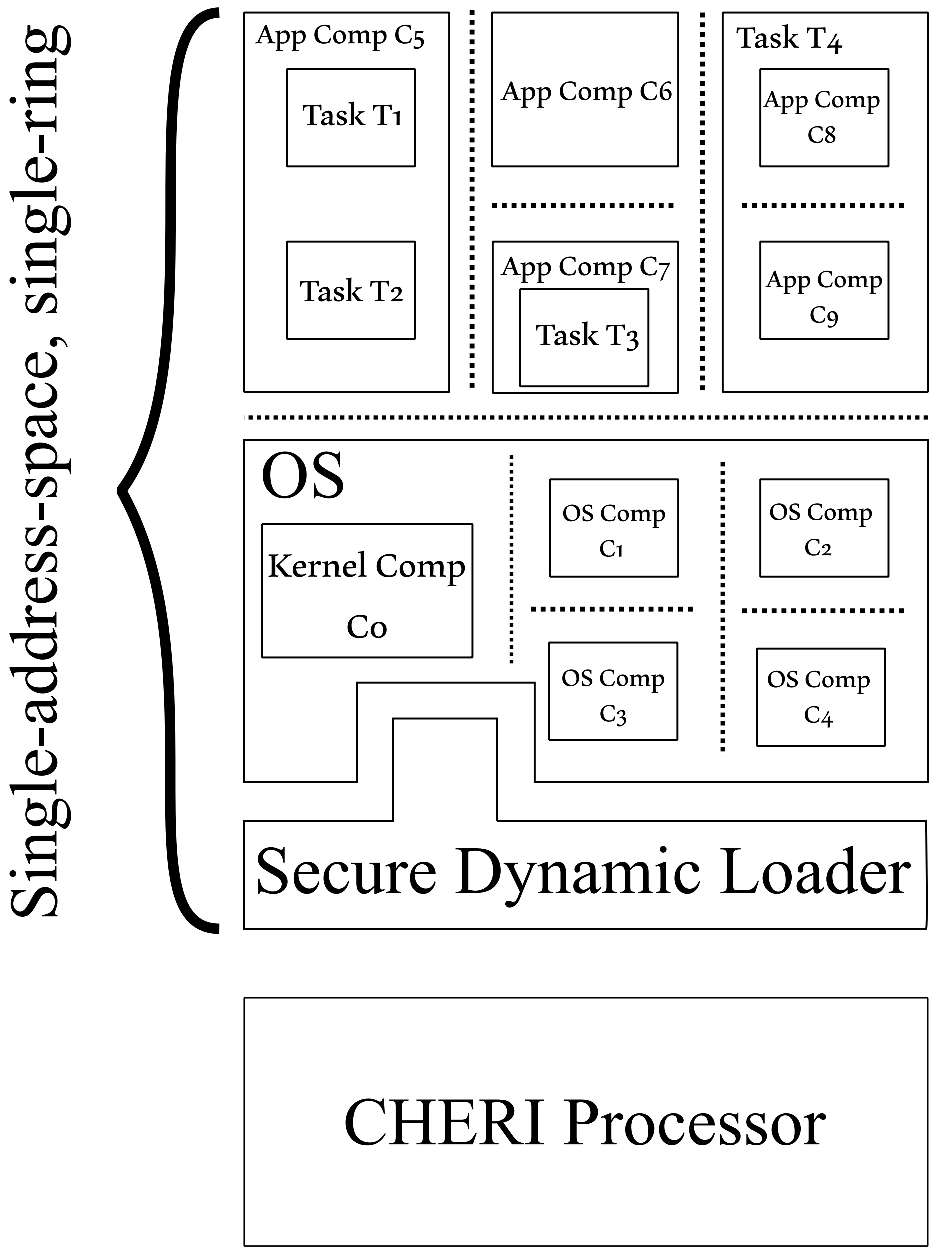


Figure 3.2: CompartOS runtime model example. The CHERI processors and the secure dynamic loader, are trusted. Everything else is untrusted, including the toolchain. Linkage module compartments are in red, and OS tasks (or threads) are in yellow. Different CompartOS' compartmentalisation models are shown as discussed in Section 3.2. Dotted lines indicate isolation boundaries with potential bridges for communication between compartments as specified by the security policy. The secure loader notch indicates some form of integration with the OS (e.g., threading).

is specified at development and compile time and enforced at runtime load and link time. Linkage-based compartmentalisation has several benefits over other compartmentalisation models described in Chapter 2:

- Intuitive: Compartments are directly mapped to source-code files, objects, and libraries.
- Compatible: No need to reimplement or redesign existing software projects.
- OS-independent: No reliance on specific threading or process models, virtual memory support, or OS-enforced access control policies (e.g., filesystem-based ACL)

A compartment is a linkage-based module.

In CompartOS, the basis of a compartment is defined to be anything that could go into one or more source-code files that will be compiled into a relocatable object module or a library. This could optionally contain code (functions), data (variables), and other sections. Compartmentalisation is thus *specified* at compile-time and *enforced* at the runtime linkage stage. That is, CompartOS relies on the linkage model and modules to define the basis of compartments statically at development and compile time. Therefore, a compartment can encapsulate applications, libraries, device drivers, secret keys, software updates, etc., independent of the task or process model.

Such benefits make it easy to compartmentalise projects like static baremetal single-threaded ones, to rich EOSes with multi-threading support, dynamic memory, and filesystems.

An inter-compartment security policy is defined by the language/linkage model.

Since compartments are directly mapped to source-code files, it is natural to make use of the programming language and linkage model to specify:

- API: to define the visibility of each symbol to other compartments.
- Communication: to define the relationship with other compartments.

The specification is later on enforced using CompartOS and a CHERI-aware secure loader.

A compartment owns static linkage-based and dynamic resources.

Besides statically defined linkage-based resources, a compartment can also own dynamic resources it allocates at runtime. Furthermore, compartments can own privileges to access IO memory regions and certain privileged instructions, defined and granted as CHERI capabilities with associated restricted permissions.

Capability-based access control.

In CompartOS, any access to resources has to be performed via a capability. Each compartment has a capability table that contains CHERI capabilities to access linkage-based and dynamic resources. The access control is specified at both language-based and OS/linkage-based compartmentalisation levels. A secure CHERI-aware compartment loader enforces both memory-safety within a compartment (like in memory-safe languages) and spatial compartmentalisation guarantees between compartments like in capability-based microkernels.

CompartOS provides a mechanism for compartmentalised fault isolation and recovery, not for catching specific security violation faults.

While CompartOS is benefiting from CHERI's memory safety for catching security violations, CHERI's security guarantees are not part of the model as a new contribution. That is, CompartOS does not require or enforce a particular mechanism or policy for catching specific security violations (e.g., buffer overflows, use-after-free, filesystem access control, etc.). It is up to the embedded systems designer to deploy and implement mechanisms for catching security-related violations and triggering architectural faults for them. For example, deployed mechanisms could be using CHERI, code instrumentations that trigger hardware assertions, remote attestation, or further hardware and software mechanisms. Instead, CompartOS' contribution is mainly focused on splitting up an existing monolithic software system into smaller compartments. An architectural fault, by whatever mechanism is deployed, is thus only associated with and confined for the affected compartment and does not affect the entire system. This enables further software fault handling actions (see Section 3.3) to be performed on the faulting compartment.

A compartment does not trap to perform a protection domain switch.

Unlike MMU and MPU based compartmentalisation techniques, CompartOS does not have the notions of system calls or user, kernel, secure, unsecure, privileged or unprivileged. Compartment switches are normal function calls in the same privilege ring and do not incur further microarchitectural overheads due to traps or hardware reconfiguration.

No software monitors, hypervisors, or background checks.

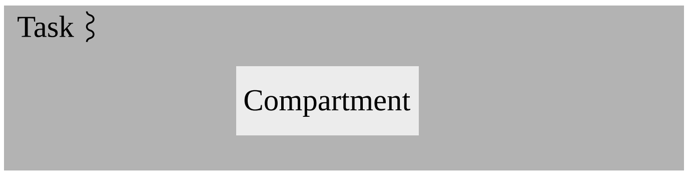
Unlike hypervisors, microkernels, and secure EOSes, CompartOS only requires a secure loader to enforce the compartmentalisation policy at load or boot time. Once compartmentalised, no further background checks or monitoring happen in software. The integrity and confidentiality of the system are maintained by virtue of capability-based access control and CHERI hardware.

CompartOS' linkage-based compartmentalisation can also support task-based compartmentalisation.

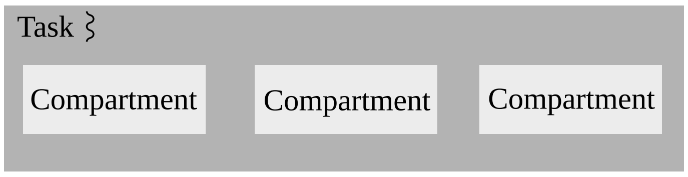
The CompartOS model could be used to support different types of compartmentalisation models as in Figures 3.3:

- **Task-based**: Single task per compartment.
- **Library-based**: Multiple compartments per task.
- **Multitask-based**: Multiple tasks per compartment.

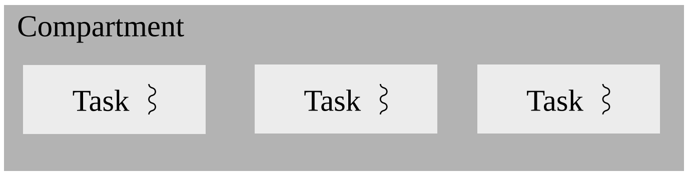
Task-based is a typical task-based compartmentalisation approach, similar to process-based compartmentalisation in UNIX and MPU-based compartmentalisation in some secure embedded EOSes. The task is the core unit of compartmentalisation. This could be realised by CompartOS by placing all of the task's code and data in a single linkage module or library and preventing any external access to its symbols. Communications between compartments are mainly task-oriented and IPC-based (e.g., message-passing or



(**a**) Task-based compartmentalisation model within CompartOS' linkage-based compartmentalisation.



(**b**) Library-based compartmentalisation model within CompartOS' linkage-based compartmentalisation.



(**c**) Multitask-based compartmentalisation model within CompartOS' linkage-based compartmentalisation.

Figure 3.3: Possible compartmentalisation models in CompartOS.

shared memory). Compartment switches happen when task context switches occur, which change capability tables implicitly (as the root capability table address is held in a CGP, a general-purpose capability register, as we discuss in Chapter 4).

Library-based has different libraries isolated and compartmentalised within a single task (or without threading support at all). This is convenient for static single-threaded baremetal embedded systems (without OS or threading support). But also, this is vital in rich embedded systems designed around modules or third-party libraries rather than tasks. For instance, the OTA demo that we evaluate later (see Section 5.13) has a single application task with third-party cryptography, SSL/TLS, OTA, and communication libraries from different companies. Compartment switches between compartments in the same task happen as wrapped function calls to external compartments that do the compartment switch.

Multitask-based could have multiple tasks in the same compartment. Communications between tasks could be IPC-based. For example, a TCP/IP stack compartment could have different threads for Ethernet driver handlers and higher-level protocol-related threads for the platform-independent parts of the TCP/IP stack.

A compartment switch changes the root capability table.

Whenever a compartment switch happens, the root capability table changes. Capability tables can change explicitly in library-based compartmentalisation by a trampoline wrapping an inter-compartment function call or implicitly in task-based compartmentalisation where two tasks are running in two different compartments, and a context switch changes the capability table (by restoring CGP).

All compartments execute in a single-address-space, MMU-less, single-privilege-ring environment.

Except for CHERI, there is no reliance on specific hardware (e.g., MMU, MPU, privilege rings) or OS (virtualisation, paging, address spaces, filesystem access control) features for protection or privilege separation. CHERI enables enforcing software compartmentalisation in a single flat physical address space. Coarse-grained user and kernel separations are not required as CHERI could isolate privileges for memory and system registers; thus, a single processor ring or privilege mode is sufficient. This makes it possible for OSes, libraries, device drivers, and applications to be all in the same privilege ring as different compartments.

Dynamic runtime compartmentalisation with actively malicious compartments.

While formal verification, type-safe programming languages, certification, and static analysis tools provide guarantees reasoning about the security of a software system, adding new features, libraries, or refactoring some existing code requires considerable effort to maintain such guarantees. Furthermore, static guarantees are obtained against certain types of known models, vulnerabilities, and exploits. Our approach, on the other hand, allows for dynamic security, including creation, isolation and deletion of compartments during (boot, load, and unload) runtime. This helps protect against unknown future vulnerabilities and exploits. Moreover, once deployed, we preserve the ability to selectively patch and update specific components if needed without affecting the remaining system.

3.3 Fault Handling

In this section, we discuss fault handling within the CompartOS model. We first specify the type of faults the model is concerned about, then list a few example mechanisms how to handle those faults in a generic, compatible, and flexible way along with the trade-offs. Depending on the type of fault, the compartment, and the fault handling technique, the handlers could involve the user application, the OS, and/or the secure loader. The evaluation and implementation of those techniques are discussed later in Section 5.14.1 within a safety-critical use case.

3.3.1 Fault Types

CompartOS is only concerned with faults that end up triggering architectural traps in the hardware. For example, those could be CHERI faults, hardware assertion faults, or bugs such as divide-by-zero or unaligned accesses faults. Such faults may have a well-defined framework for fault handling in general-purpose OSes, but that is not the case in most embedded OSes. For example, the UNIX process model has the signalling mechanism [117] for delivering faults to users; an MMU fault delivers a *SIGSEGV* [118] signal to the faulting process, which the user could optionally register a handler for; otherwise, the process will be terminated by default. On the other hand, few embedded operating systems have a process model or a signalling mechanism, and they do not even consider handling architectural faults. For example, neither RTEMS nor FreeRTOS has support for handling hardware exceptions; instead, they halt the entire system. They rely on extensive testing and the assumption that such faults will not occur in favour of performance, smaller code size, and simplicity. As Compa has availability as one of the main requirements, fault handling was an intuitive thing to support.

The software could react to faults differently in a fine-grained way, depending on the cause of the fault. Some faults require no actions, while others may need more serious handling, such as micro-reboots. While CompartOS could handle any architectural faults, we only focus on faults that are security and safety related. That is, we do not discuss faults that may require emulations (e.g., trap-and-emulate). Examples are mapping an MMU page on faults or emulating unaligned memory access. Further, some MPU-based compartmentalisation solutions trap-and-emulate MPU accesses as a workaround for not having enough hardware memory regions to isolate resources per compartment. CompartOS does not have or attempt to emulate any of such faults as there is no hardware limitation on the number of resources per compartment to protect. In the following section, we categorise the main fault types handled by CompartOS.

CHERI faults can be categorised into spatial memory-safety faults, capability delegation faults, access-system-registers faults, and user faults. Such faults prevent exploits from happening in the first place, and this maintains the confidentiality and integrity of the system. The causes of the faults could be a malicious compartment, code bugs, or wrong programming assumptions about privileges (e.g., ambient authority). It is important to notice that CHERI faults reflect software misbehaviour that might or might not represent an attempted exploit. CHERI faults can be detected in:

- debugging scenarios, to identify errors at runtime; this is similar to static (e.g., CHERI compilation warnings) and dynamic analysis techniques (e.g., runtime testing trying to trigger faults). Fixes could be applied at this stage before going into production.
- production runtime scenarios, where vulnerable compartments or third-party compartments exist (e.g., dynamically loaded applications or libraries). Those can be malicious (e.g., trojans) or have bugs. Further, compartmentalisation helps to defend against future unknown vulnerabilities in this case.

Thus, faults encountered in debugging scenarios could be fixed easily, while in production, it might be more challenging and need more work to handle them properly, which is the focus of this section.

For CHERI spatial memory faults, it might be enough to return an error to the caller compartment, to which it can take further actions. This is similar to a SIGSEGV signal sent to the user process when it violates MMU memory security in UNIX OSes. For example, a fault in *memcpy* could simply return an error code to the caller, and no further actions are needed. This may require changing the API of memcpy and require developers to write code defensively by checking return error codes, which is discussed later. Other CHERI faults such as access-system-registers could indicate an actively malicious compartment that might require a more strict response, such as killing a compartment (if not safety-critical) or restarting it.

Assertion faults are triggered when the software is written more defensively to catch logical, algorithmic errors, unexpected or wrong user input values, or just a something-is-wrong and stop technique. The aim here is to stop the program from executing further and potentially help debug the issue. Non-safety-critical projects could simply stop the entire system and print an error message on assertions. However, in safety-critical systems that require reliability and availability, this might not be an option. If the faulting compartment (due to an assertion) is not safety-critical by itself, but the rest of the system is, it might be enough to kill that compartment and deny further access to it. If the faulting compartment itself needs to be available, more fault handling techniques might be applied, such as restarting it in a known good state or implementing custom fault handlers that can correct the error and support continued execution.

(Micro)architectural faults include unaligned accesses, illegal instructions (e.g., floating-point), divide-by-zero, trap-and-emulate, etc. Those almost certainly need special handling if they are expected to happen. For example, unaligned and trap-and-emulate faults could have an emulation library to do the proper functionality. If the faulting compartment is not expected to fault on such traps, other handling techniques might be applied depending on the compartment requirements (criticality, trustworthiness, etc.). While our demonstrations of CompartOS focus on safety and security related faults, CompartOS supports registration of arbitrary fault handlers, so designers of high-availability systems would likely register handlers for this class of fault.

3.3.2 Fault Handling Mechanisms

We experimented with four main fault handling strategies that fit with our design goals. There are trade-offs between them when it comes to compatibility, robustness, simplicity, determinism, and assumptions. The CompartOS model does not restrict systems to use any, but gives flexibility as much as possible to meet specific embedded system requirements set by the system designers. We start with language/API and UNIX-signal like mechanisms that require interventions and some development efforts by the developers. Thus, they may not be the best option from a compatibility perspective, but they could contribute to a robust and secure system. Those mechanisms involve error checking, custom (user) fault handlers, and defensive programming. Then we discuss two more fault handling mechanisms that are more compatible from users' or application developers' perspective; they are part of the secure loader or the OS and are similar to default actions for UNIX signals (e.g., to terminate a process), but they mainly deal with linkage modules instead. We call these mechanisms *Compartment-Kill* and *Micro-reboot*.

3.3.2.1 Return-Error

Returning an error code in a function-call manner to the caller compartment is the simplest fault handling technique. It is assured that the timing of the error handling will take the very minimum amount of time compared to other techniques and hence meets most real-time requirements and is the fastest. However, it does not address any side effects or implications that may have taken place from the time the compartment was entered up until the fault. For example, a TCP/IP compartment might receive a packet, allocate buffers for it, send it to higher-level protocols layers and then fault. The allocated buffers, in this case, will be leaked. An attacker could use that knowledge to cause heap exhaustion and DoS. Similarly, if a compartment holds a lock, then faults and returns without releasing it, that might cause a deadlock or DoS. The caller compartment might also benefit from an implementation that always checks for return error codes, but this needs some knowledge about compartment boundaries and some development efforts or good programming practices not to assume or rely on the callee function having performed its service properly. This could hurt compatibility, especially for badly-written existing software not following good error checking practices, but with a minimum effort, it could be worth the determinism and performance advantages.

Return Error fault handling scheme is good for cleanly-written existing software that does error checking and for simple compartments that are almost stateless and do not expect to often fault.

3.3.2.2 Defensive Programming

If compatibility is not a major requirement, some good programming practices could be used for compartments written for CompartOS to reduce faults and minimise the overheads of other fault handling techniques that we discuss next. Still, the development overhead is not as disruptive as rewriting the entire software in another more secure language (e.g., Rust) or porting applications to a new OS like in microkernels. Writing programs defensively in CompartOS is left to the programmer and is not part of the model or our evaluation. However, we discuss it here for completeness.

Error checking, especially across compartment calls, is the simplest yet most convenient defensive programming technique. It requires an understanding of the compartment boundaries, APIs, error codes, and how to handle them. Fault details can then be completely forwarded to the compartment itself as an error code, as opposed to a fault handler. This is similar to try-catch programming techniques. Return-Error fault handling discussed earlier could then be directly used with the most performance advantage. Further, paying attention to warnings (specifically CHERI ones) and fixing them, besides other static analysis tools, could prove vital in minimising faults and costly fault handling techniques.

Dealing with dynamic memory (i.e., heap memory) might be challenging from embedded systems perspective due to its complexity and non-determinism. Some embedded systems completely rely on statically allocated memory at build time. Whenever possible, this eliminates lots of complexity and potential vulnerabilities associated with dynamic memory allocation and revocation. However, in cases where heap allocation is required, more care might be needed. For example, if a compartment is to be killed, its dynamically allocated memory might also need to be freed and/or zeroed. This requires bookkeeping of compartments and their dynamically allocated memory and iterating over them on revocation, which may result in code overhead and non-determinism. Even when that is not an issue, some care must be taken when killing a compartment that allocated a memory region and handed it over to another compartment. In that case, other compartments having this derived memory region should be denied access to it, or the killing process should fail until no other heap memory regions are being used by other compartments (i.e., by maintaining a reference counter). Such solutions could be fine for general-purpose systems, but in most embedded systems, they are not. Thus, it is left to the developer to implement those solutions rather than the CompartOS model itself.

Most of the commonly used memory-related functions in libc, such as *memcpy*, *memset*, *memzero*, *memmove*, *strcpy*, etc., could also be refined to avoid exceptions in the first place. Since in CHERI purecap mode, those functions all receive pointers as capabilities with base and bounds, they can be modified to check the valid bounds and return an error code. This maintains compatibility from user respective while avoiding potential costly out-of-bounds CHERI exceptions. Further, tags could also be checked before accessing the buffers to ensure that the passed pointers are valid, which also prevents CHERI faults due to invalid forged pointers. Similarly, the programmer should always design the code to pass bounds on buffers and check for them whenever possible before dereferencing them.

Finally, the compartmentalised system should be designed and implemented assuming that it will fault, rather than just trying to prevent faults. Based on each compartment's requirements and the remaining system, a proper fault handling technique can be chosen at design time.

3.3.2.3 User-level Custom-Handler

A custom fault handler allows developers to register fault handlers per compartment. This is similar to microkernel-style fault handling (e.g., in seL4) or a handled UNIX signal (e.g., SIGSEGV) that the user registered a custom handler for. It is the responsibility of the developer to implement the proper logic to maintain the overall system integrity and availability and handle any further required actions to meet custom requirements. While being the most flexible and robust technique, it assumes complete knowledge of the underlying compartmentalised system along with the CompartOS implementation as well. The system has to be redesigned or reimplemented to assume faults are going to occur and handle each fault differently. Hence, the compatibility requirement is ruled out. As CompartOS is linkage-based, a custom handler can be specified as a function per compartment with a specific name. Then, during the loading process, the secure loader automatically searches if this specific function exists and registers it as a callback function. For instance, in Section 5.14.1, a TCP/IP custom fault handler is called *CheriFreeRTOS_FaultHandler*, which is just a function. CompartOS, as a model, allows registering and calling custom fault handlers. However, the implementation and behaviour of the handlers are completely compartment and OS-specific, and are not part of the model.

Custom-Handler is a good technique for systems that are designed around CompartOS from scratch or for mainstream systems that can tolerate additional development and maintenance efforts to meet specific requirements of the system and not necessarily others.

An example of this is restarting the TCP/IP stack in FreeRTOS. This requires deep knowledge of the TCP/IP stack, including its state, the threading model, resources allocated by the stack, events, and any dependent compartments so that it sends them further signals. In the ECU demo (see Section 5.14), we implement a full restart as a custom handler that resets the state, frees the buffers, deletes tasks before sending a signal (OS-specific) to the ECU application as a dependent compartment. The ECU application has to also be redesigned, assuming faults would occur from callee compartments and expect signals accordingly for each compartment. When a restarting TCP/IP stack sends a signal to the ECU application, new sockets have to be created instead of the killed ones, and the application has to call the TCP/IP stack initialisation function, which also restarts the Ethernet driver. In the same use case, the ECU application compartment itself also has its own custom fault handler which, on specific faults, frees allocated buffers and keeps working.

Many safety standards (e.g., ARINC 653 standard [119]) already provide the infrastructure necessary for the full support of CompartOS handlers. They already require a monitor and a recovery strategy table, so the assumption is that they already have code to recover from the different possible faults. We can just perform a one-for-one replacement with CompartOS custom handlers. This might greatly simplify the work of implementing safety standards in a CompartOS implementation (e.g., CheriFreeRTOS), including any qualification and certification efforts. For example, we can model a partitioned system around CompartOS that complies with the ARINC 653 standard [119], which is used for safety-critical real-time avionics. Such model can be implemented on top of CheriFreeRTOS. While CompartOS enforces the ARINC 653 “space partitioning” requirement, it can also support flexible fault handling techniques implemented by another “health monitor” subsystem, also defined by the standard. The health monitor subsystem is responsible for detecting faults and recovering from them, with a recovery strategy table for each compartment that can be directly mapped by CompartOS customer handlers.

3.3.2.4 Compartment-Kill

Killing a compartment follows the UNIX-style of sending a kill signal to a faulting process that does not have a registered fault handler. This is the default behaviour for most UNIX signal types that get triggered due to faults. The main difference is that, in CompartOS, a linkage module is killed rather than a process. Still, depending on the compartmentalisation model as discussed in Section 3.2, a task that is attached to a linkage module could still be killed or suspended in a task-based compartmentalisation model in CompartOS. The idea is to stop the compartment from executing and deny or fault on compartment entry attempts. Further, resources owned by a killed compartment can be invalidated and released if necessary. Compared to resetting the entire system on a fault (like what traditional EOSes do), killing a compartment that is not safety-critical could improve availability. For example, killing a music player in an automotive self-driving car instead of resetting the entire system is a more desirable action. This further helps prevent exploits or attacks on the faulting compartment, which could be in the form of DoS attacks that continually force a system reset.

This scheme does not require knowledge or interaction from developers and is rather generic; thus, it is good for compatibility requirements. It is more secure and robust than the return-error scheme as it can invalidate and release any owned resources. It works well if the to-be-killed compartment is not safety-critical and the requirement is to keep the remaining system available and functional. However, this technique comes with a performance and determinism cost compared to the return-error one, especially if there are attached dynamic resources.

Compartment-Kill is thus good for relatively complex applications with less safety-critical compartments that are willing to pay a performance cost for compatibility and security purposes. The system designer should also be aware of any further implications and dependent compartments chain that will be denied services by the killed compartment.

An example is Amazon's OTA application discussed in Section 5.13; a failed or disabled software update compartment is not often critical, but killing the compartment that holds secret keys might be required. Another example is a shell or a user interface in automotive or avionics (as in Section 5.14); such compartments can be killed so that they do not affect the steering and brakes compartments that could be life-threatening.

3.3.2.5 Micro-reboots

Micro-reboots is a design and an implementation of [73] but for CompartOS and embedded systems. It aims for maintaining the availability and compatibility for both the faulting compartment (unlike the Compartment-Kill scheme) and the remaining system. The faulting compartment is rolled back to a known working state regardless of the error. In most cases, this does not require the intervention of developers and thus meets the compatibility requirement. However, implementing this technique might be relatively costly when it comes to memory consumption, power, and performance. Still, it is better than doing a full reboot or killing a compartment, especially if it is a safety-critical one.

Some implications might need to be considered. For example, once the reboot process starts, no other compartments can call into or reference the faulting one until it is completely restarted. This could be implemented by immediately killing the compartment on faults then performing the reboot. A more robust but less compatible solution is to use the underlying OS' signal and task mechanisms to notify dependent compartments about the reboot state.

Micro-reboots could be a good recovery strategy for systems that do have enough memory space to maintain snapshots of compartments state and sections, while compatibility and availability (of the faulting compartment) are the main requirements. An example application where micro-reboots could be useful is Amazon's OTA. Software updates themselves do not have strong real-time or performance requirements, but the focus is to perform secure and reliable updates. Without being able to do micro-reboots in OTA (due to a faulting or malfunction compartment), the entire OTA process might fail completely, putting the system under further security risks.

3.3.2.6 Conclusion

We have discussed different fault types and techniques that CompartOS is designed to handle. While being able to support those mechanisms is part of the model, the implementations will greatly vary depending on the OS, applications, the compartments, the programming languages, and the runtime. Thus, we do not attempt to provide concrete fault handling models but just abstract ones to demonstrate the flexibility of CompartOS. Still, we provide evaluation and implementation details of those mechanisms in Section 5.14.1 within a realistic, real-time, and safety-critical use case in FreeRTOS. We also trade-off compatibility, performance, and robustness of such models within embedded systems.

3.4 Summary

In this chapter, we described an abstract CompartOS model that could be applied to a variety of embedded (operating) systems. We first defined the requirements and goals of the model, and how they differ from state-of-the-art solutions. Then, we illustrated the core design principles of CompartOS and how they meet the requirements and goals. At the end of the chapter, we walked through fault handling trade-offs in the embedded systems field with a focus on the requirements we described earlier. We adopt some of the discussed fault handling mechanisms in this chapter and implement them in the next chapters to evaluate the flexibility of CompartOS when it comes to fault handling.

CHAPTER 4

CHERIFREERTOS: AN IMPLEMENTATION OF THE COMPARTOS MODEL

In this chapter, we discuss how we implement the CompartOS model spanning three different software components: a *toolchain*, a *dynamic loader*, and an *embedded operating system*. These parts collaborate to carry a compartmentalisation policy from compilation and code generation through to runtime using static and dynamic linkage of a set of modules. The toolchain needs to be able to build separate linkage modules whose ABIs enable partitioning the system. The dynamic loader needs to be CompartOS-aware in order to create proper capability-based protection domains at runtime, depending on the security policy. The Embedded Operating System (EOS) needs to cooperate with the dynamic loader in areas such as attaching a compartment to a thread, fault handling, and dynamic memory allocation. I extend the existing CHERI-LLVM toolchain, libdl dynamic loader, and the FreeRTOS EOS to implement the CompartOS model. The remainder of the chapter explores this design and implementation in detail.

4.1 Implementation Strategy

To evaluate the CompartOS model, I have implemented these three subsystems in the CHERI-LLVM toolchain, the libdl dynamic loader, and the FreeRTOS embedded operating system. These existing software artefacts are part of the CHERI project and already support CHERI's memory protection model, but not software compartmentalisation. For simplicity, I refer to my implementations of those three software subsystems collectivity as *CheriFreeRTOS*. CheriFreeRTOS is thus suitable for evaluating changes in compatibility, scalability, security, and performance, which will be described in Chapter 5. Figures 4.1 and 4.2 illustrate the affected components and how the model is mapped into CheriFreeRTOS implementation.

Toolchain First, the toolchain in Figure 4.1 is to an existing CHERI-LLVM toolchain; I have extended it to support a new GPREL addressing ABI for CompartOS and discuss it in Section 4.2.

Dynamic loader Second, the dynamic loader in Figure 4.2 is libdl (described in Section 2.8), which I have extended to support the loading and linking process of CompartOS-aware compartments built with my modified CHERI-LLVM toolchain. libdl enforces a security policy at (boot) runtime. Section 4.3 describes how I have extended the baseline libdl to support CompartOS' software compartmentalisation.

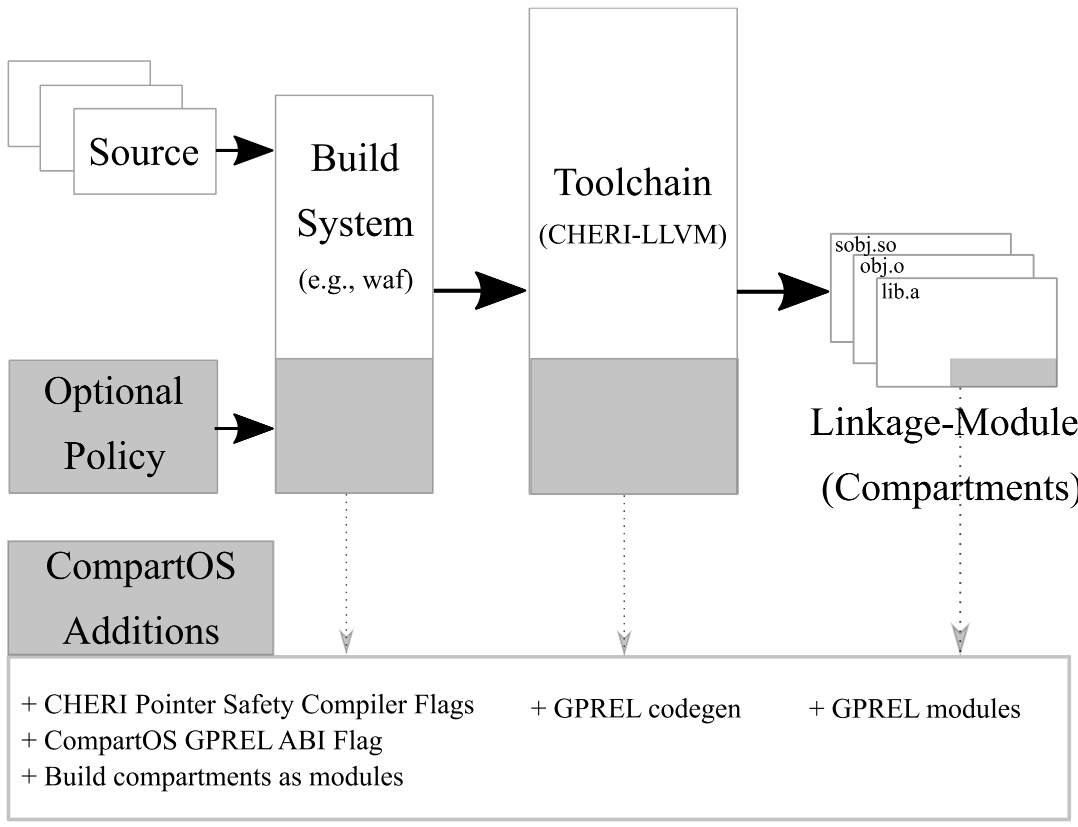


Figure 4.1: CompartOS build-time workflow. Grey boxes indicate the CHERI/CompartOS additions I have made to existing components and tools. Developers feed the (legacy) source-code to the build system as well as an optional security policy that splits up the source-code into logical compartments with well-defined APIs (e.g., libraries), along with the restricted communications between them. The output is linkage modules that are fed to the dynamic secure loader at runtime.

Embedded Operating Systems Third, the EOS in Figure 4.2 is FreeRTOS, to which I have added a few modifications and integrations with libdl to provide software compartmentalisation, specifically in the threading and fault handling support. FreeRTOS modifications are described in Section 4.4. Section 4.4.4 walks through a compartmentalised CheriFreeRTOS use case from the evaluation chapter (see Section 5.12). It briefly describes what can be placed into a compartment, by example. The use case walks through a scenario of receiving an Ethernet packet, performing domain crossings from the lowest level FreeRTOS exception handler and moving across a device driver, a TCP/IP stack, and an FTP server.

Finally, in Section 4.5, I discuss the security, compatibility, and performance implications and limitations of some implementation decisions I have made in CheriFreeRTOS and how to approach enhanced security implementations as future work.

4.2 CHERI-LLVM: New GPREL Capability Addressing ABI

Current embedded operating systems, even the existing CHERI-based ones, compile and build statically and get linked into a single binary image that executes in a single address space and single privilege ring. This means that all code can call any other code, and access any other

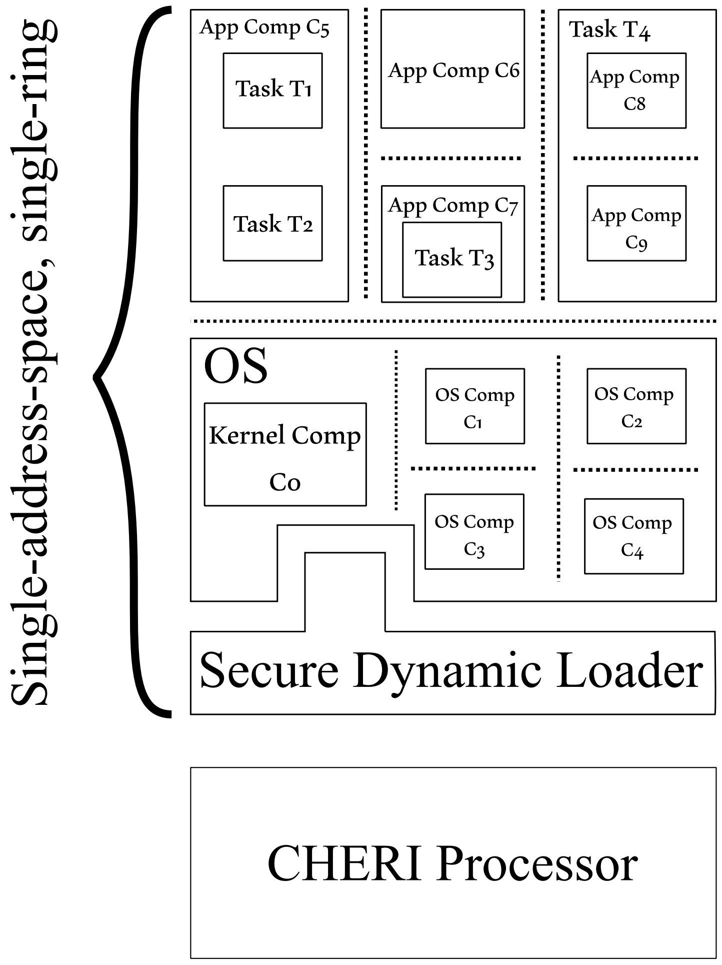


Figure 4.2: CompartOS runtime model example. Green boxes, including the CHERI processors and the secure dynamic loader, are trusted. Everything else is untrusted, including the toolchain. Linkage module compartments are in red, and OS tasks (or threads) are in yellow. Different CompartOS' compartmentalisation models are shown as discussed in Section 3.2. Dotted lines indicate isolation boundaries with potential bridges for communication between compartments as specified by the security policy. The secure loader notch indicates some form of integration with the OS (e.g., threading).

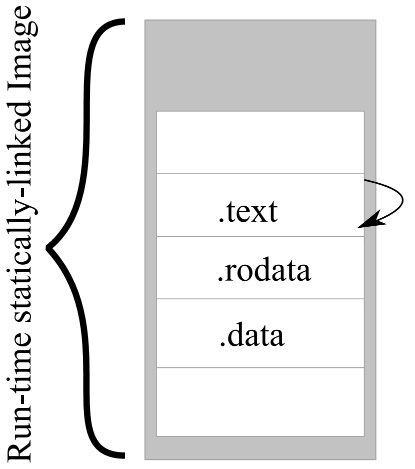


Figure 4.3: A runtime example of a function call in a baseline, statically-linked, and loaded FreeRTOS ELF image. The image contains all the software content (kernel, drivers, libraries, and applications) like code and data stored into a contiguous physical memory without any isolation or protection. An external function call directly loads (using PCREL addressing) and jumps to the function address in step #1.

data. Global variables and functions can thus be referenced from anywhere. For example, an application can call a kernel function that is not part of the kernel's API and services. An example of a baseline FreeRTOS system that performs a function call is shown in Figure 4.3. In this system, the kernel, device drivers, applications all get linked and loaded into a single contiguous physical area of memory. Global variables usually get stored in the program's *.data* section, while functions get stored in the *.text* section. Static linking works fine without compartmentalisation, but in order to carry around compartment policy via linkage, some form of dynamic linking needs to be supported.

Compilers and linkers try to optimise references to global variables and functions by addressing them relative to the current program counter (PC). This addressing is usually called PCREL and is used whenever possible in embedded systems if the RISC architecture can offset a variable from the current PC. In contrast, compilation and linkage in general-purpose OSes are different. Applications work in a separate address space from the kernel and cannot call it directly; instead, they use system calls that require support by the OS and an underlying processor with at least two privilege rings. Further, referencing shared user libraries (as discussed in Section 2.7) is Global Pointer (GP)-relative and requires OS features such as MMU virtualisation, lazy binding, and dynamic linking.

When compiling for the pure-capability CheriABI [33] UNIX environment, the CHERI-LLVM toolchain currently uses PCREL addressing. This does require the PC capability register (PCC) to cover the entire address space in order to be able to offset any area of memory, including required captable load operations. Consequently, functions cannot be bounded and can access anything in its single-address-space environment (i.e., a UNIX process, with statically-linked binaries in CheriABI). At build time, the CHERI-LLVM toolchain generates a single static captable for the whole ELF image, as shown in Figure 4.4, which gets populated at program startup. Though semantically different, CHERI purecap mimics Global Offset Table (GOT) and (Procedure Linkage Table) PLT models as in shared libraries in general-purpose systems by adding an extra layer of indirection between loading resource addresses and accessing them. However, GOTs and PLTs are largely not needed and not used in embedded systems.

Having a single globally shared captable that is PCREL-accessed suffices for UNIX-based

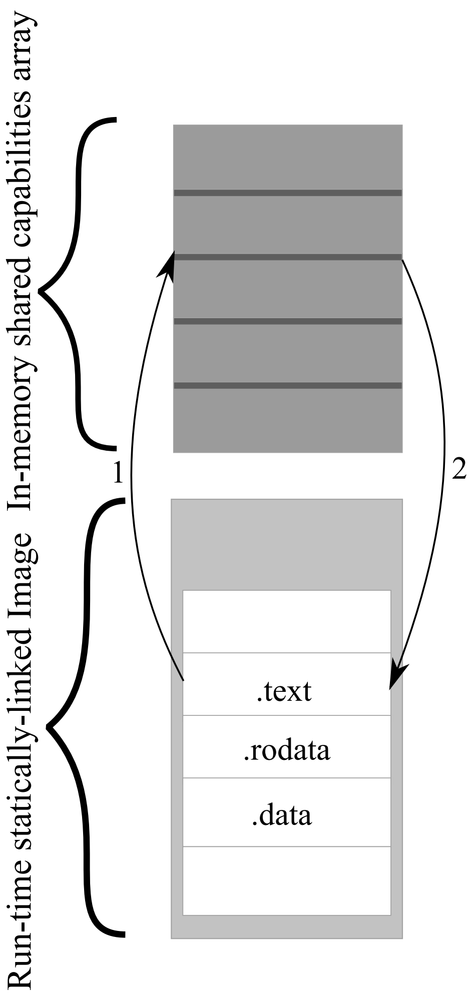

In-memory shared capabilities array
Run-time statically-linked Image
1
2
.text
.rodata
.data

Figure 4.4: A runtime example of a function call in a non-compartmentalised, CHERI purecap, statically-linked and loaded FreeRTOS ELF image. The image contains all the software content (kernel, drivers, libraries, and applications) like code and data stored into a contiguous physical memory. Spatial memory protection and pointer safety are enforced by CHERI to protect against memory safety violations. A single shared and global captable is created at boot time, and any references to globals have to load (using PCREL addressing) a capability first (step #1) before dereferencing it (step #2). However, applications can still access all kernel's globals from the captable without restrictions using PCREL addressing.

```
void *capability;

void  cap_reference(void) {
    capability = (void *) 0;
}
```

Listing 4.1: C code referencing a capability

```
cap_reference:
cincoffset      csp, csp, -32
csc             cra, 16(csp)
.LBB0_1:
auipcc   ca0, %captab_pcrel_hi(capability)
clc      ca0, %pcrel_lo(.LBB0_1)(ca0)
csc      cnull, 0(ca0)
cret
```

Listing 4.2: Generated CHERI-RISC-V assembly code referencing PCREL capability

systems with well-defined process models and MMUs, but is inadequate to support compartmentalisation in embedded systems with a single physical memory space. For example, if this PCREL addressing was employed in EOSes with CompartOS, it would create a single captable granting all compartments (e.g., applications, drivers, kernels, libraries) unrestricted shared access to all globals. CompartOS, therefore, requires runtime support for a single-address-space image with multiple independent captables and bounded functions (e.g., that cannot address anything relative to their address, except their code) as shown later in Figure 4.7.

We overcame this challenge by adding new support for per-compartment captables; however, this required changes to the ABI. Specifically, we used another architectural capability register, the Capability Global Pointer (CGP) and changed the compiler ABI for compartments to generate GP-relative (GPREL) instead of PCREL accesses. Each compartment's CGP holds its root captable. Listings 4.1, 4.2, and 4.3 show an example of C code accessing a capability and its generated PCREL and GPREL assembly code, respectively. ***cap*** is a pointer capability accessed by the code. Listing 4.2 shows PCREL addressing in the first three assembly instructions in its `.LBB0_1` block. Listing 4.3 shows GPREL addressing, which (at present) requires extra instruction in its `.LBB0_1` block. During runtime loading of a compartment, `libdl` fixes up the GPREL relocation in the first three instructions with the proper offset of the installed capability.

```
cap_reference:
cincoffset  csp, csp, -32
csc         cra, 16(csp)
.LBB0_1:
lui         a0, %captab_gprel_hi(capability)
cincoffset  ca0, cgp, a0
clc         ca0, %captab_gprel_lo(capability)(ca0)
csc         cnull, 0(ca0)
cret
```

Listing 4.3: Generated CHERI-RISC-V assembly code referencing GPREL capability

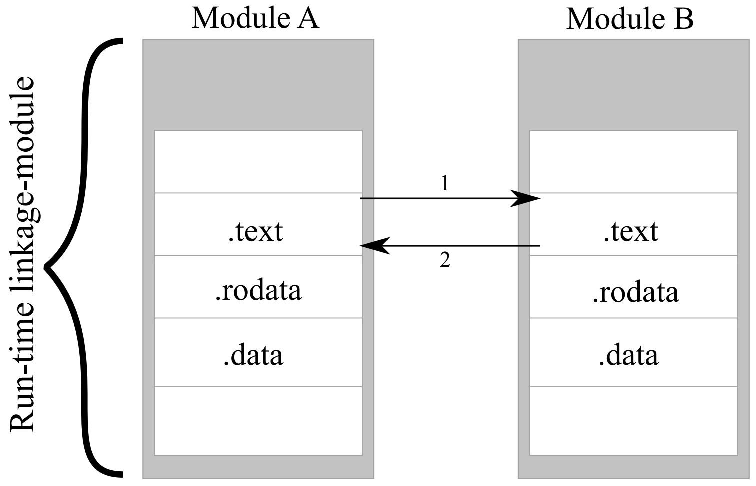


Figure 4.5: A runtime example of a function call in a baseline libdl of two dynamically linked and loaded linkage modules. The linkage modules are dynamically loaded by libdl and stored into non-contiguous regions of memory (dynamically allocated), still without any protection and isolation between them after the final systems is completely loaded. The final runtime image contains all the software content (kernel, drivers, libraries, and applications), loaded as linkage modules, without any isolation or protection. An external function call directly loads (using PCREL addressing) and jumps to the address in step #1, using PCREL addressing.

Within a compartment, any capability addressing is now GPREL instead of PCREL. Allocation and construction of captables are done by `libdl` dynamically at runtime and discussed next in Section 4.3. The outcome is a dynamic system where each function, variable, and symbol has a specific address, length, and permissions.

4.3 libdl: Adding CHERI Compartmentalisation to a Dynamic Linker and Loader

As discussed in Section 2.8, the original `libdl` (a dynamic linker and loader library) provides no protection between linkage modules such as objects, libraries and the base ELF image (e.g., the OS kernel), and in fact, has no mechanism it could even use to do so. That is, once new objects or libraries are loaded, the resulting system is indistinguishable from a baseline system statically linked at build-time. Figure 4.5 shows how, in runtime, object modules can call global external functions in other modules. This is considered the baseline for a dynamically loaded system by libdl. Even if we build libdl and its loaded modules in CHERI's purecap mode, there will only provide spatial pointer memory-safety, but not isolation and compartmentalisation between loaded linkage modules, which can still access a globally shared captable between modules as shown in Figure 4.6.

I have extended `libdl` to support CHERI and CompartOS linkage-based software compartmentalisation, which is a powerful composition:

- `libdl` defines linkage modules such as ELF objects and libraries. Thus, we use its linkage model and implementation as a baseline for our CompartOS model by building on libdl's linkage modules as a basis for compartments.

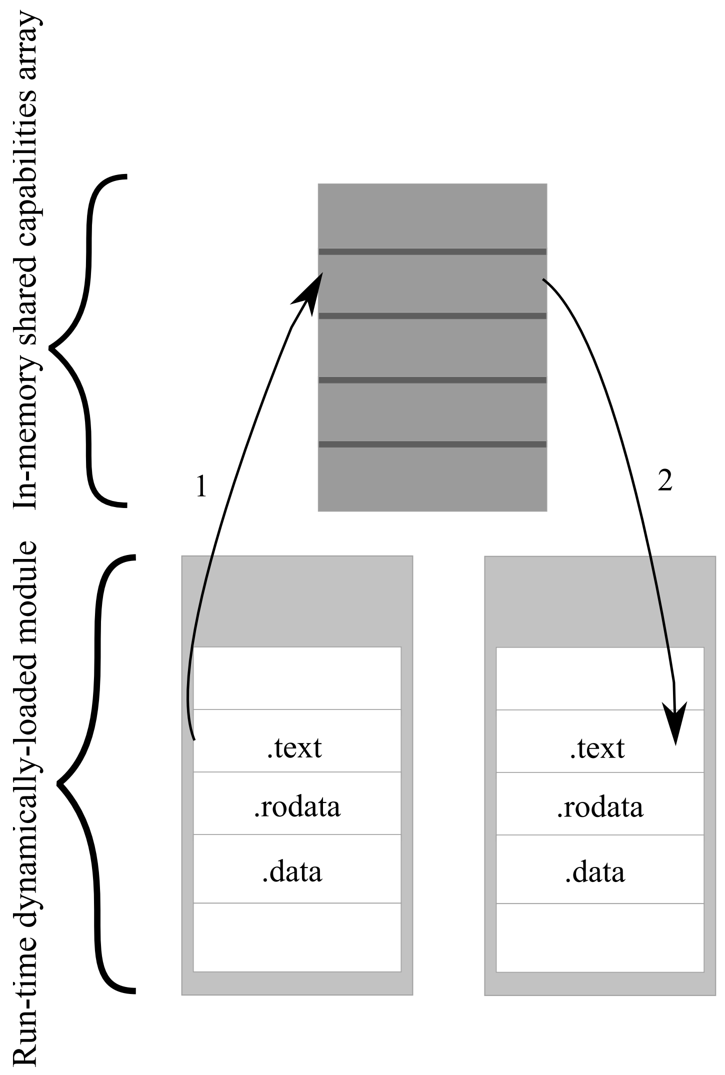

In-memory shared capabilities array
Run-time dynamically-loaded module
1
2
.text
.rodata
.data
.text
.rodata
.data

Figure 4.6: A runtime example of a function call in a non-compartmentalised, CHERI purecap, dynamically-linked and loaded FreeRTOS systems with libdl. The final loaded system contains all the software content (kernel, drivers, libraries, and applications) like code and data dynamically allocated and stored into a non-contiguous physical memory. Spatial memory protection and pointer safety are enforced by CHERI to protect against memory safety violations. A single shared and global captable is created at boot time, and any references to globals even between different modules have to load a capability first using PCREL addressing (step #1) before dereferencing it (step #2). However, all loaded modules can still access each other's and all kernel's globals from the captable without restrictions using PCREL addressing.

- CHERI provides lightweight fine-grained memory protection and software compartmentalisation using capabilities. While CHERI offers memory safety for libdl, CompartOS extends libdl to implement and enforce the three compartmentalisation models: task, library, and multi-task, models discussed in Section 3.2.

libdl acts as a secure boot loader or a trusted firmware. That is, it is the most privileged piece of software that first takes control of the hardware on reset, holds access to the CHERI root capabilities (i.e., initial capabilities that hold all privileges such as PCC and DDC), and then loads and grants compartments the minimum restricted capabilities they need to perform their job.

In the following sections, we define what a compartment is, describe how a compartment is created by libdl from the linkage model perspective, and illustrate how a compartment switch performs a protection domain switch.

4.3.1 Compartment Definition

A compartment is a dynamic runtime entity that gets created out of a static ELF object file or a library (i.e., linkage module). A compartment is strictly associated with one capability table that defines its entire protection domain and gets dynamically allocated and created at load time according to a security policy. Besides the captable, a compartment also has an ID and a fault handler. As discussed in Chapter 2, a protection domain is a logical unit of isolation with privileges attached to it to access resources and optionally communicate with other compartments. Figure 4.7 shows an example of two runtime compartments in a CheriFreeRTOS compartmentalised system. The only way a compartment can access external resources is through its captable, by loading a capability from there.

4.3.2 Compartment Creation

A compartment is dynamically created by calling *dlopen()* with the path to the module in the file system. After `libdl` allocates the memory necessary for loading and linking, a capability table is also allocated with an initial size of the defined symbols count within that module. External function symbols are treated as inter-compartment domain switches. Those are detected by `libdl` during a compartment creation.

For every external (i.e., inter-compartment) function call within a compartment, an additional lightweight function trampoline is emitted. The implementation of that trampoline is discussed later in Section 4.3.3. Capabilities can be minted (by libdl) from other capabilities through monotonic action. If an inter-compartment call is allowed (by some user-defined policy), `libdl` mints a capability from the callee's captable and installs it in the caller's captable dynamically. Figure 4.8 briefly shows the process of creating a compartment. Darker boxes indicate CompartOS' compartmentalisation actions. The captable will come to hold capabilities for four different classes of linkage-based symbol capabilities:

- **Local Capabilities:** Local C symbols (e.g., global variables and functions), accessible only by the module defining them. They cannot be minted or called from other modules.

- **Global Capabilities:** Global C symbols, accessible by the module defining them. They cannot be directly minted.

- **Interface Capabilities:** A subset of global capabilities from which others can be minted if the security policy permits (see Figure 4.1).

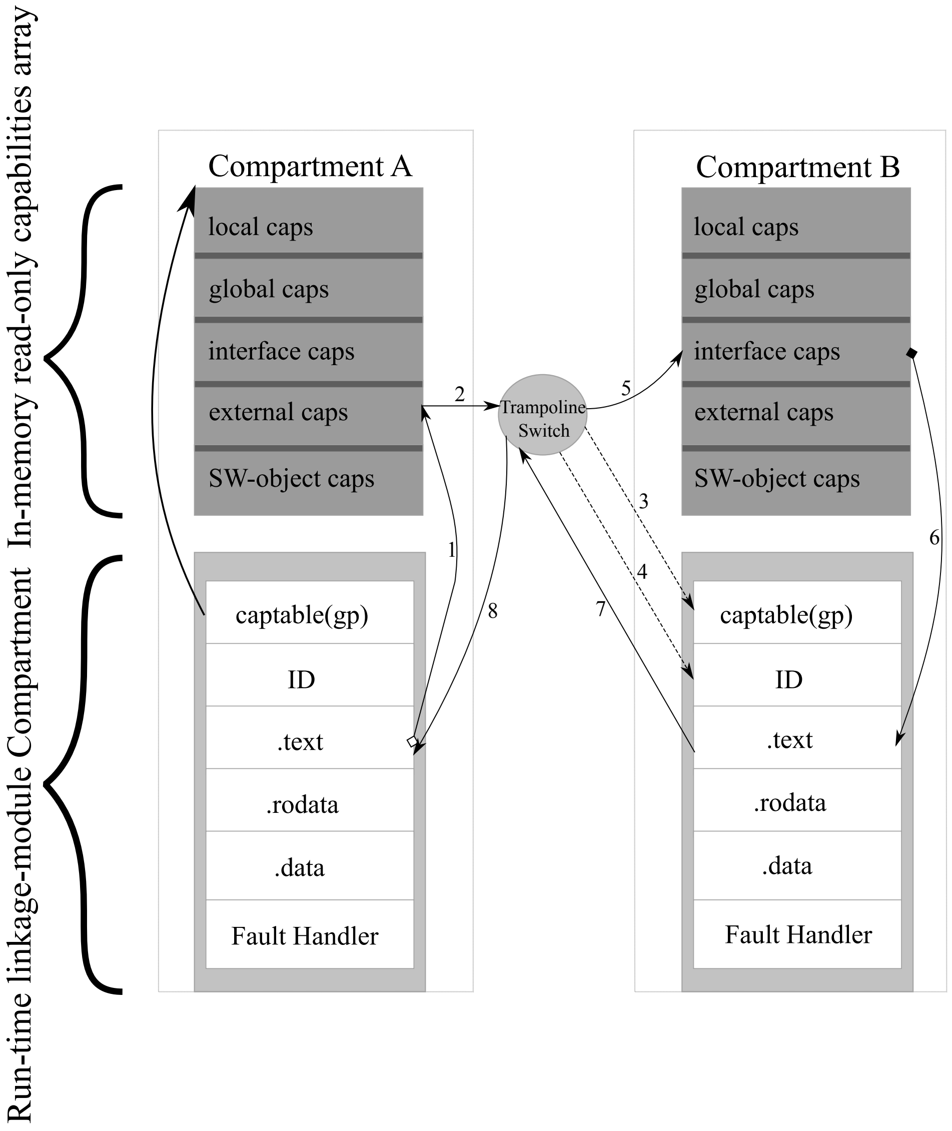


Figure 4.7: A runtime example of a compartmentalised system and a protection domain switch. Filled lines are capability references, while dashed lines are operations. Numbered edges denote the sequence to perform an external call to another compartment, which triggers a domain switch. The switch starts with a function call that references a capability from the externals capability list (#1). If the external capability is present and valid, it points to a small, read-only trampoline (#2) that performs compartment switches by setting the new captable and compartment ID (#3 and #4) after storing the caller's context. It then jumps via the interface capability (#5) provided by the target compartment. This interface capability points to the function within its associated compartment (#6). Upon its return (#7), the trampoline restores the caller's context, captable and ID, then returns back to the caller function (#8).

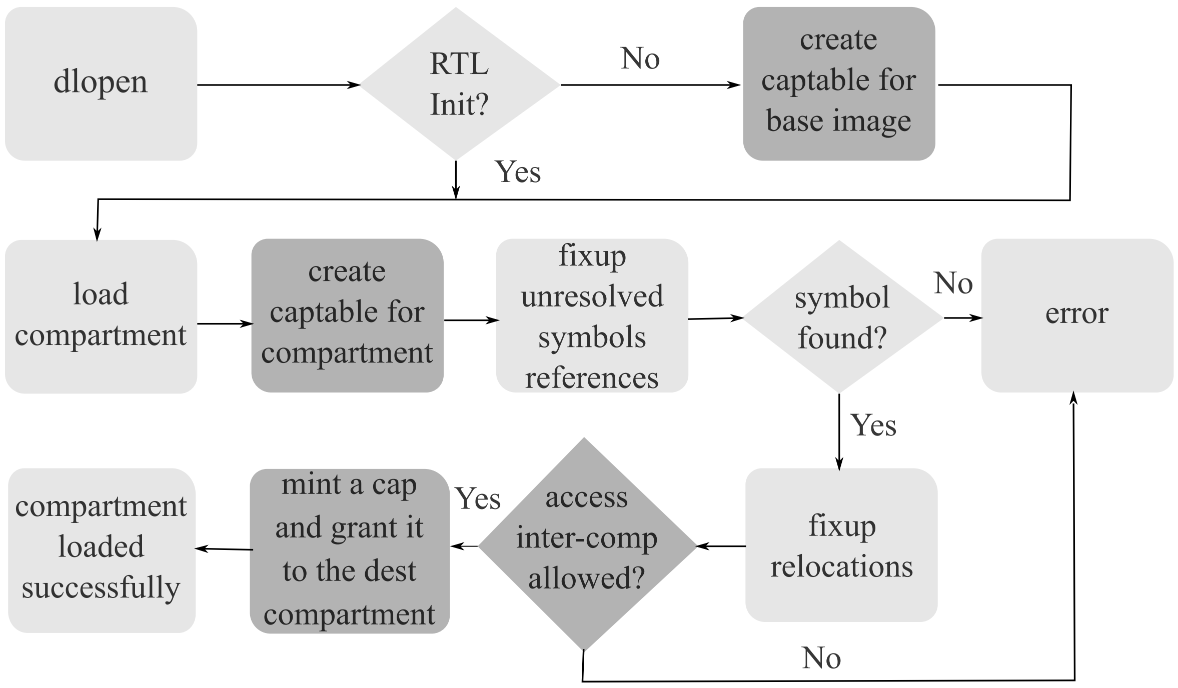

dlopen
RTL Init?
No
create captable for base image
Yes
load compartment
create captable for compartment
fixup unresolved symbols references
symbol found?
No
error
Yes
compartment loaded successfully
mint a cap and grant it to the dest compartment
Yes
access inter-comp allowed?
fixup relocations
No

Figure 4.8: CompartOS compartment creation process. Darker boxes indicate CompartOS' compartmentalisation actions.

- **External Capabilities:** Capabilities to external resources that are minted and granted (possibly with stricter permissions) to this compartment.

On boot time, libdl automatically loads and links all the compartments that the embedded system designer designates in a configuration file, as in Figure 2.3. The configuration file is considered part of the security policy. In Figure 2.3, the system has a configuration file called *libdl.conf* which defines 5 ELF static library compartments and one ELF object module as an application. libdl automatically creates, mints, and partitions the captable according to the security policy as in Figure 4.1. The default security policy we deploy in our implementation is language and linkage-based, as discussed in Section 3.2. For example, a library compartment such as *libc.a* has global variables and functions, but a specific API (e.g., open, close, read, and write functions) that are also global and visible. Other compartments are only allowed to access the API, but not other global variables. On load time, libdl will create a captable for *libc.a* with all of those globals and default their linkage visibility to *hidden*. Then, libdl would iterate over the API functions of *libc.a* that the system developer defines (part of the security policy) by marking their linkage visibility as *visible* and creates a set of interface capabilities out of them in the captable. On *foo.o* load time, libdl will mint a *write* function capability from *libc.a* to *foo.o*, if *foo.o* tries to call *write*. Any other non-API global functions from *libc.a* that *foo.o* tries to access will be denied and not minted, and a security violation would be returned during the loading process of *foo.o* and consequently *foo.o* will not be loaded.

CheriFreeRTOS aligns with C and ELF linkage-based symbols attributes. For example, it creates read-only capabilities for code and constants. Similarly, stack capabilities and *malloced* memory (from compartments) do not have execute permissions. Capabilities can also refer to software resources (e.g., from the kernel compartments) and not just memory regions and ELF-based symbols. Besides ELF-based resources, each compartment has a table of runtime

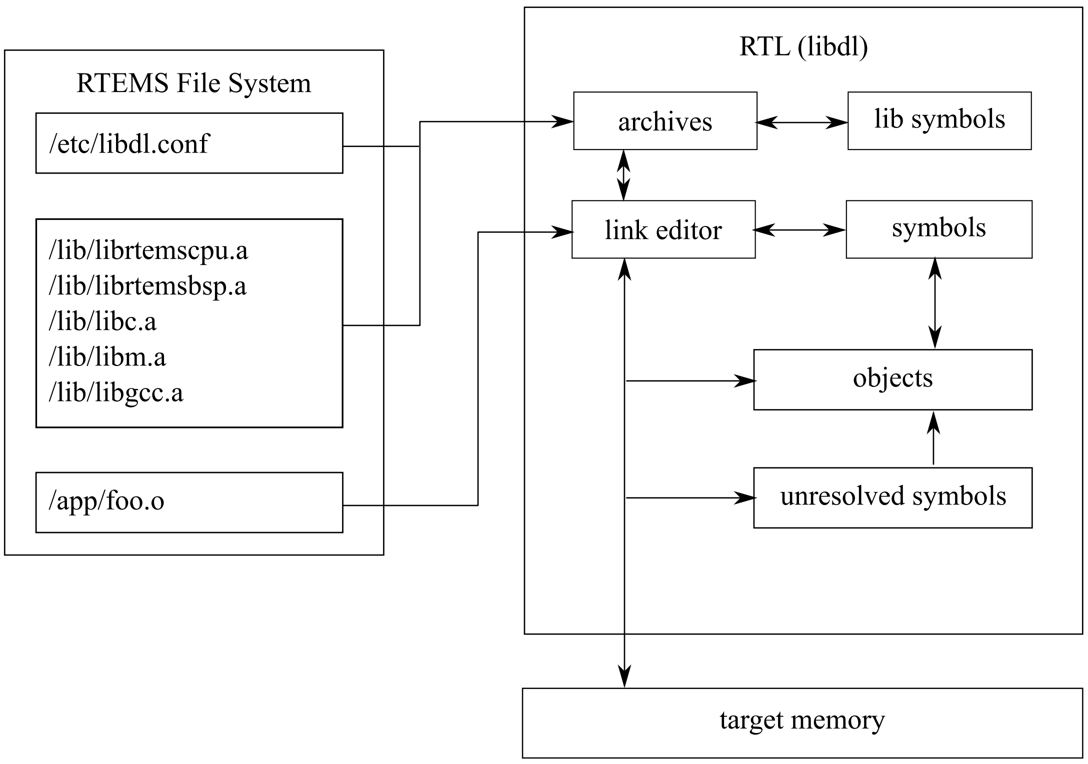


Figure 4.9: LIBDL: Dynamic Loading and Linking Process in RTEMS. The picture is copied with permission from the libdl author and maintainer, Chris Johns. The picture source is from [80].

software capabilities pointing to resources that it creates. Example software resources are tasks, queues, or timers in FreeRTOS. This allows for more object-capabilities like in a microkernel style, such as kernel objects or user-defined resources.

4.3.3 Compartment Switch

A compartment switch triggers a protection domain switch between distrusting compartments, as shown in Figure 4.7. As discussed in Chapter 2, a protection domain switch is a transition from one protection domain with specific privileges to another with different privileges. `libdl` detects inter-compartment calls at load time and emits a read-only, *Sentry* trampoline that performs a compartment domain switch accordingly. *Sentry* (sealed entry) functions are CHERI capabilities that can only be jumped to but not read from or written to. Additionally, `libdl` wraps function pointers with a trampoline since those can be leaked between compartments. The trampoline consists of metadata and code to define which compartment a function belongs to, the destination compartment captable, and a bounded capability to the function itself (see Listing 4.4). `libdl` sets up the metadata (function capability, destination captable, and compartment identifier) during compartment loading. As mentioned before, a trampoline is a read-only sentry function that can only be jumped to by external compartments but not read from or written to. The only privilege it has is the metadata.

```
 xPortCompartmentEnterTrampoline:
% Metadata
.Lfunc:        .zero CAP_SIZE % Callee's function capability
.Lcaptable:    .zero CAP_SIZE % Callee's capability table
.Lcompid:      .zero CAP_SIZE % Callee's compartment ID
// Compartment/Domain Switch
Save caller's context (register set, GP, compid)
Setup new GP/captable
Set currentCompartmentID = Lcompid
(Optional) Restrict/bound the callee's stack
Call the destination function
Restore caller's context (register set, GP, compid)
Return to the caller compartment
```

Listing 4.4: Compartment trampoline pseudo code

4.4 FreeRTOS: Integrating Linkage-based Compartmentalisation in an Embedded OS

In this section, we describe the subsystems where the dynamic linker and an embedded OS may need to cooperate. The required change in an EOS to support linkage-based software compartmentalisation is its threading support. Optionally, if the EOS supports dynamic memory allocation, some changes might also be required there. In the evaluation chapter (see Section 5.9), we further describe, in detail, the exact numbers of line of code changes per a FreeRTOS subsystem to support CHERI's memory safety and CompartOS' linkage-based software compartmentalisation.

4.4.1 Compartments and Threading

A thread or a task, in the context of EOSes, is a unit of execution, with its own stack, that can be scheduled. In CheriFreeRTOS, a thread can enter exactly one compartment at a time, and compartment entrances can be nested in a function-call manner. A compartment is also re-entrant from different threads, assuming the function is re-entrant and thread-safe. When a thread enters a compartment, it donates its remaining stack to the compartment. The bookkeeping is handled by the FreeRTOS `TCB_t` structure, which is extended to contain a compartment context, including the current compartment ID (xCompID), the caller's return address, and the caller's return stack.

4.4.2 Resources and Ownership

There are five broad categories of resources in CheriFreeRTOS that get accessed via capabilities:

- C-based memory objects: Functions, global variables and pointers.
- IO memory regions.
- Dynamic memory: Dynamically allocated memory at runtime (e.g., using *malloc*).
- Kernel-based objects: Such as tasks, queues, and timers.

- Software objects: Higher-level resources and data structures, such as sockets, TLS contexts, and encryption keys.

C-based objects and IO regions are owned by the compartment that defines them. The ownership of such resources is statically known at compile-time, and capabilities are created for them during compartments loading. The other three resource categories are created later after boot time. They are owned by the compartments that call the API functions that create them. For example, a compartment that calls *malloc* owns the returned region of memory. This also applies to created tasks, queues, or sockets. Ownership restrictions can be further enforced by limiting what API functions a compartment can call. Besides the captable created at loading time, a capability list is also created per compartment for runtime resources.

4.4.3 Software Fault Handling

I have modified FreeRTOS to support handling architectural exceptions for the RISC-V port. The baseline port does not handle or support any form of exception handling and instead hangs or performs a reset instead. Exceptions go through a few stages:

Architectural exception is the lowest (hardware) level which interrupts unprivileged execution and often traps to the most privileged hardware ring (if there are at least two rings) and software subsystem (e.g., firmware, hypervisor, or an OS). It requires privileged access to the processor's configuration to acknowledge and clear register flags, for example. Once done, it forwards some exception details to a higher-level piece of software such as an OS.

OS fault message could be, for example, an OS (or a subsystem of it), which composes a richer *fault message* that is both hardware related and OS related, describing the cause of the fault and the affected components at a more generic and standardised fault model implementation.

Software fault handler Finally, the OS could optionally forward the fault to a custom higher-level *software fault handler* which could exist in a per-compartment handler, or a user signal handler such as in UNIX signals, for example.

4.4.3.1 Fault Handler Registration

As discussed in Section 3.3, there are different types of fault handlers in the CompartOS model. Those can involve either libdl, FreeRTOS or both of them. In CheriFreeRTOS, a custom fault handler can be registered per compartment by defining and implementing a *CheriFreeRTOS_FaultHandler* function for each compartment at development and build time. During the loading process, libdl searches for any occurrence of *CheriFreeRTOS_FaultHandler* as a function symbol and registers it as a callback function for the loaded compartment. Otherwise, if libdl could not find *CheriFreeRTOS_FaultHandler*, it applies default fault handling techniques such as Return-Error, Compartment-Kill, or Micro-reboot if configured and enabled by the system designer at build time.

4.4.3.2 Runtime Fault Handling

If an exception occurs, the modified FreeRTOS architectural exception handler checks if the current task is running in a compartment (by checking the associated per-task compartment structure), and if so, it jumps to a registered custom per-compartment software fault handler (if provided). In the case of Return-Error and Compartment-Kill, the exception handler returns back to the trampoline that performed the compartment switch, which returns to the caller compartment with an optional return error code.

These per-compartment software fault handlers improve fault tolerance and maximise availability. In Section 5.14.1, I have experimented with and evaluated several fault handling techniques to explore their overhead, complexity, and robustness trades-offs. Techniques employed include simply returning to the caller with an error code, resetting some state of a compartment, or performing a completely isolated software update and (micro-)reboot of a library, such as TCP/IP.

4.4.4 Examples of Compartments in CheriFreeRTOS

There are multiple software subsystems that can be compartmentalised in CheriFreeRTOS. For example, device drivers, higher-level exception handlers, OS libraries, and applications can all be compartments. Figure 4.10 is an example of a compartmentalised use case that we evaluate later in Section 5.12. We walk through a scenario where an Ethernet packet is received. First, the most privileged low-level software is typically exception and interrupt handlers. This performs the lowest level operations required by the processor hardware, such as acknowledging and clearing interrupts (typically privileged operations), before they call higher-level handlers that typically exist in device drivers. For example, in Figure 4.10, an Ethernet interrupt will first interrupt an existing application code and execute in the *cherifreertos* kernel before further switching to the compartmentalised *virtio* library that contains a device driver. The compartment switch there happens via a *trampolined* function call as the *virtio* exception handler is registered with FreeRTOS as a callback handler (i.e., function pointer) in its interrupt vector table. The *virtio* driver manages and configures the Ethernet device, for example, by performing DMA operations and writing to buffer rings. Once done, the *virtio* compartment performs a compartment switch to an even higher-level compartment called *freertos_tcpip*, which implements and handles higher-level internet protocols. The compartment switch here happens over an IPC by which the *virtio* task queues a request over FreeRTOS' queues to the *freertos_tcpip* compartment. Finally, a compartment switch to an application server (e.g., FTP) will be performed, depending on the type of the received packet.

4.5 Security Implications, Limitations, and Future Work

I have considered the implementations of protection domain crossing, threading, and stacks that embrace functional correctness, compatibility, scalability, availability, threat model, and performance, when enforcing compartmentalisation. Compatibility makes it possible to adopt the model in almost all EOSes like in CheriFreeRTOS without changing their threading model design choices and assumptions. I consider CheriFreeRTOS as a first step that allows compartmentalising a monolithic embedded OS into logically separate (i.e., spatial compartment safety) compartments that allow recovery from spatial CHERI's memory safety violations (besides others), without having to largely refactor and redesign the source-code.

Next, we discuss some implementation decisions I have made that are security-related, with potentially enhanced implementations as trade-offs that can be future work.

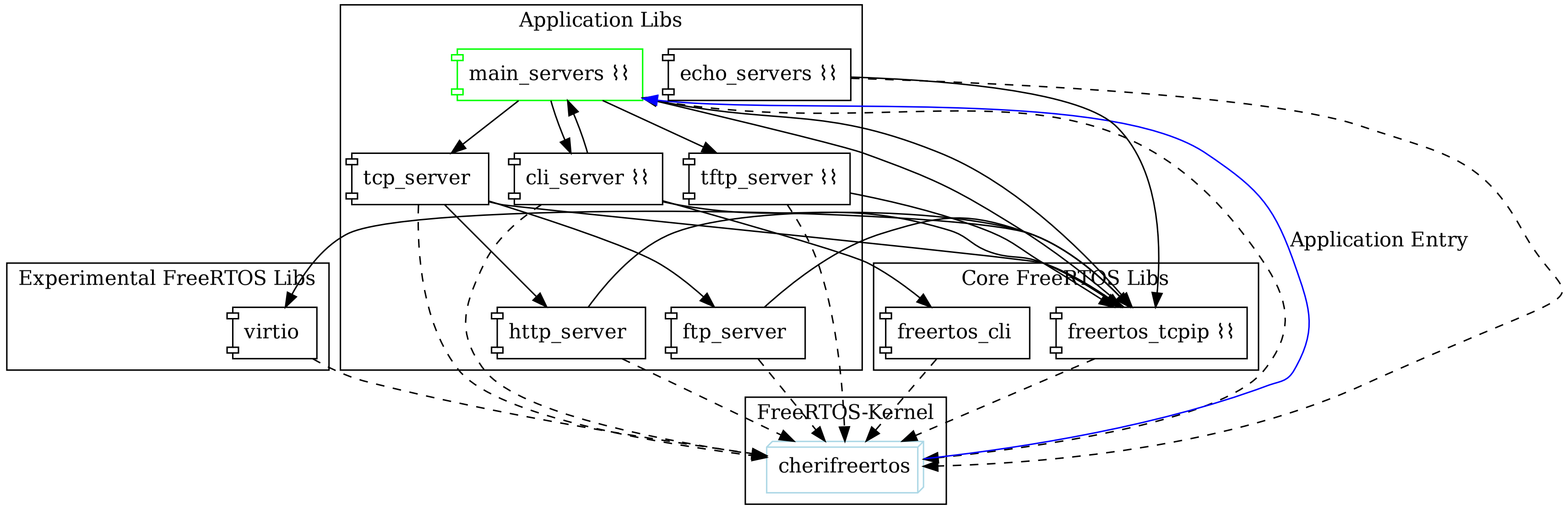


Figure 4.10: Compartmentalised TCP/IP demo graph that is statically and automatically generated by the CheriFreeRTOS build system during the build stage. Small boxes represent linkage-based library compartments, and the edges are the API references a compartment makes to another. Dotted edges indicate a trusted call to FreeRTOS without trampolines or domain switches. Compartments are placed into larger boxes to categorise them into core, third-party, and application libraries.

4.5.1 Concurrency

Concurrency is managed by the developer, not CompartOS or CheriFreeRTOS. We do not try to manually manage concurrency at the software compartmentalisation implementation, but we leave that to the developers to design a thread-safe code at the OS and application level, e.g., by using event-based designs or fine-grained locking on entering a library compartment that modifies shared global states. For example, initialising the FreeRTOS TCP/IP compartment changes a global variable to indicate the TCP/IP subsystem has already been initialised and running. If two threads try to initialise the TCP/IP stack at the same time, a race condition might occur. It is possible to wrap a compartment entry (e.g., in the trampoline) with a FreeRTOS lock on entry, for example, to avoid that. However, this is considered an OS and an application-level issue rather than a linkage-based compartmentalisation issue. Further, library entries (e.g., via function calls) can be cyclic and trigger indirect recursive entries, which could cause deadlocks if CompartOS decided to blindly wrap compartment entries with locks. CheriFreeRTOS tries to maintain compatibility by being agnostic to the (third-party) library implementations. We found that FreeRTOS is well designed and implemented when it comes to thread safety, and it required no further concurrency changes to the source-code or compartmentalisation subsystem. This should be the common case for most other EOSes as well.

4.5.2 Trusted Computing Base

In our CheriFreeRTOS implementation, I have made a decision to trust the kernel (besides libdl), as that is the common case in embedded operating systems. libdl is not responsible for handling interrupts and exceptions after it has already compartmentalised the system, and it is not expected to be interrupted at boot time. It does not act as a secure-monitor or a hypervisor, as we discussed in our model (see Section 3.2). It is the job of the FreeRTOS kernel part of its low-level exception handling, which requires more processor and hardware privileges. Thus, the FreeRTOS kernel could gain more privileges over an interrupted low-privileged application compartment before it might jump to a software fault handler in a device driver compartment (see Section 4.4.3).

Hence, I have chosen not to emit trampolines or perform compartment switches from applications or OS libraries to the *cherifreertos* kernel (see Figure 4.10). Instead, applications and OS libraries simply perform function calls via CHERI's sentry capabilities to the kernel's API. Further, I also build the kernel with PCREL addressing as it is trusted.

However, trusting the kernel might not be the ideal approach for secure systems where applications may not trust the OS kernel. In this case, the FreeRTOS kernel itself can be built as a normal compartment with GPREL addressing and calls to it will be trampolined protection domain switches. Further, the lowest level of architectural exception handling can also be moved out of the FreeRTOS kernel to a compartment. This can be achieved by libdl by modifying the trampolines to context switch the CHERI-RISC-V's *mtcc* capability register (machine trap code capability) and attach each compartment with a different *mtcc* register (similar to different CGPs). In CHERI-RISC-V, *mtcc* holds a code capability that is jumped to (by the hardware) on any interrupt or exception. This completely removes the kernel from interrupting a compartment and gaining privileges over it.

4.5.3 No Temporal Safety

CompartOS can be implemented in various different ways with design and implementation space, requirements, and trade-offs that we cannot fully cover because of the wide diversity in embedded systems and, in particular, their (security) requirements. As discussed in Section 3.2, our compartmentalisation model is not much about the containment of malicious code as much as containment and isolation of crashes and recovering from them. For instance, using CHERI as a security mechanism for catching memory-safety security violations has already largely addressed the issue of catching malicious and arbitrary code execution (i.e., by triggering architectural exceptions on memory-safety security violations), while CompartOS takes one step into partially recovering from them. Supporting temporal safety defends against attacks such as use-after-free, double-free, and potentially leaked capabilities between compartments over trampolined compartment switches (as arguments or shared stacks) or implicitly when (re)using (re)allocated memory buffers. While temporal safety is an important security feature, I have made an implementation decision in CheriFreeRTOS not to support it for the purpose of this thesis as it is not one of our requirements and is not covered by our threat model. Our threat model already assumes the existence of malicious or vulnerable compartments (e.g., third-party or supply-chain libraries). Further, as discussed in Section 3.1, our main design goals and requirements are compatibility, scalability, availability, and functional correction when compartmentalising a mainstream embedded software system. Those are different requirements from state-of-the-art systems that may support temporal safety (e.g., uVisor and Rust/TockOS); most of them do not have or support compatibility, scalability, and availability as main requirements. That said, other CompartOS implementations or future work can support temporal safety if it is a major requirement. This will be considered as a further security mechanism for catching temporal safety faults, and it is the responsibility of the system designer to implement and deploy it (see Section 3.2).

Examples of improving temporal safety in CheriFreeRTOS (as future work) may include zeroing non-argument capability registers on compartment entry, zeroing the callee compartment stack on return, or completely dedicating multiple stacks for each thread trying to enter different compartments (i.e., instead of having a single stack per thread, shared between the compartments it enters).

On the one hand, having the CheriFreeRTOS implementation not currently supporting temporal safety does not undermine its applicability or importance in securing embedded systems; CheriFreeRTOS can still defend and *recover* from a large number of spatial memory-safety attacks while maintaining spatial compartment isolation. In the evaluation chapter (see Section 5.12.3), we show how CheriFreeRTOS can catch and recover from 10 out of 13 public vulnerabilities in its TCP/IP stack. On the other hand, supporting temporal safety would affect our compatibility design goal; it will largely require modifications to the OS and language runtime subsystems (e.g., memory allocators, threading, stacks). Not all embedded systems support dynamic memory allocation, for example, while for those that do, changing their design and implementations might not be that tempting. Further, most embedded software systems tend to be more static in terms of memory allocation, and they rarely load/unload compartments (after boot time) compared to general-purpose OSes. Thus, supporting temporal memory safety might come at unnecessary compatibility, complexity, and performance costs, especially for less dynamic embedded systems.

4.6 Summary

In this chapter, we described an implementation of the CompartOS model that we call CheriFreeRTOS. The implementation spans over three main software components that are the toolchain (LLVM), a secure dynamic loader (libdl), and an embedded operating system (FreeRTOS). This prototype provides an example implementation of the CompartOS model. We first showed how we modified the existing LLVM toolchain to support a new capability-aware GP-relative addressing ABI that is necessary for partitioning capability tables and CompartOS compartmentalisation. We then illustrated how we extended the upstream libdl loader to be CompartOS- and CHERI- aware in order to load objects and libraries at runtime while compartmentalising them. Finally, we described how FreeRTOS and libdl are integrate in areas such as threading and fault handling, and we gave a complete example of a compartmentalised FreeRTOS-based system. At the end of chapter, we listed the limitations, implications, and future work of the CheriFreeRTOS implementation.

CHAPTER 5

EVALUATION AND RESULTS

We evaluate the CompartOS model through our prototyped CheriFreeRTOS implementation, considering its impact on performance, security, compatibility, scalability, and practicality, demonstrating fully-functional real-world use cases on hardware platforms. We also compare CompartOS against unprotected, PMP-based, and PURECAP (non compartmentalised CHERI) models and prototypes. Section 5.9 discusses the compatibility and disruptiveness when porting existing embedded OSes and applications to CHERI and CompartOS. Section 5.10 evaluates performance on the lowest level of the OS and software implementation, effectively measuring the performance of critical code paths such as context switching, compartment domain switches, IPC, and differences between protection domain switching mechanisms in both deployed and research systems. Section 5.11 is a macrobenchmark for relatively realistic and heavy workloads performing operations that are commonly used in embedded systems such as matrix multiplications, lists manipulations, and state-machine iterations. Section 5.12 evaluates TCP/IP performance which involves IO, memory, rich TCP/IP stack management, with multiple application servers built in separate compartments. In the same section, we evaluate security based on published vulnerabilities in the FreeRTOS' TCP/IP stack and how CHERI and CompartOS could help in this regard. Finally, we evaluate real-world use cases in Sections 5.13 and 5.14 and apply various fault handling mechanisms in a safety-critical application to evaluate the availability and real-time requirements in deployed scenarios. All the performance numbers and figures in this chapter are, when applicable, the medians of running benchmarks multiple times. Some figures also contain standard deviation error bars.

5.1 Evaluation Challenges

Evaluation in the field of diverse embedded systems with different hardware and software systems poses some challenges when it comes to comparing against state-of-the-art systems and related work. As discussed in Section 2.1, "embedded systems" can refer to a huge range of things from tiny microcontrollers doing power management to multicore Linux systems. Therefore, we pick a specific "size" or "scope" of embedded system to target and do not attempt to seek to address all or arbitrary embedded systems. That is, we target safety-critical, rich, complex, mainstream embedded systems in need for security and compartmentalisation without much restrictions on memory size or footprints, but still require determinism, and hence may use an MPU while running a single-address-space, single-privilege-ring embedded OS such as FreeRTOS but would not use an MMU nor support virtual memory. The following evaluation challenges persist even when we narrow down the scope of targetted embedded systems, in particular, if we compare against the evaluation and benchmarking on general-purpose systems (e.g., Linux/FreeBSD on x86/Arm):

- No present hardware-software system provides the same level of security granularity and linkage-based compartmentalisation as CHERI and CompartOS to compare against.
- Existing state-of-the-art embedded security architectures are MPU-based, without updated, published, and reproducible hardware latencies. For example, "deployed" systems such as FreeRTOS-MPU, RTEMS, TockOS, uVisor/Mbed do not evaluate performance or publish updated numbers about their critical operations, while "research" systems such as ACES, TyTAN, and TrustLite do that, but might lack some realism for adoption and deployment in industry.
- It is not realistic to compare, for example, IPC latencies between different OSes, running on different hardware platforms, with different workloads. IPC and context switch implementations are different in OSes and processors, and most have custom optimisation techniques for specific toolchains and hardware platforms. For example, an OS might support Thread Local Storage (TLS) or allow pre-emption during context switching while others do not. Similarly, an OS might apply cache partitioning techniques which may introduce extra security advantages and performance implications, which is hardware dependent. Further, there is an implicit hardware cost of software traps that largely differ between processors and is rarely reported in off-the-shelf processors (e.g., Arm). Even when building the same application workload, different toolchains might affect how the code and data are laid out in memory, which affects caches and performance.
- There are no standard/generic target-independent benchmarks for embedded systems such as SPEC [120] for performance benchmarking on general-purpose systems, IBS [121] for isolation benchmarking, or penetration testing tools for network security. More specifically, there are no benchmarks to quantify security and compartmentalisation in a generic, well-defined and agreed on way.
- None of the state-of-the-art and related work addresses availability or reports response time and recovery when it comes to faults in MPU-based compartmentalisation.
- None of the state-of-the-art and related work addresses compatibility, securing complex mainstream multi-threaded applications (like OTA).

5.2 Evaluation Methodology

We tried our best to have a realistic real-world evaluation by imitating high-end deployed systems such as the ones discussed in Chapter 2.1 and research systems discussed in Section 6 on the hardware and software fronts.

To get fair and comparable evaluations, we have implemented different variants of compartmentalisation software that include both CHERI-based and MPU-based, as well as an unprotected baseline. All are running on the same baseline OS (FreeRTOS), same workloads, built with the same toolchain (LLVM) and same baseline hardware (RISC-V softcore running on VCU-118 FPGA platform). These variants are mapped to related work and state-of-the-art systems to compare CompartOS against. We believe this is a realistic approach, and it provides accurate comparisons and overheads as if CompartOS was implemented on their original systems (e.g., Mbed OS on Arm-based processors). Further, this rules out hidden hardware, toolchain, and OS latencies/optimisations that are not relevant to the compartmentalisation and security subsystems. Finally, since embedded systems are application-oriented to perform a custom specific task, we demonstrate fully functional, deployed, application-level use cases for realism in Sections 5.12, 5.13, and 5.14. When applicable, we directly link conclusions based on the evaluations below to the hypothesis we make in Chapter 3.

5.3 Acknowledgements of the Contributions of Others

The evaluation in this chapter both depends on, and also in places considers, CHERI-extended processor designs on FPGA, as well as CHERI simulators, compilers, debuggers, and toolchains. These were developed by the broader CHERI research team, Bluespec Inc, FreeRTOS and RTEMS developers, and others and are not a contribution of this dissertation. I gratefully acknowledge those artefacts, without which this research would not have been possible.

Specifically, the hardware measurement section reports the numbers of existing development efforts by both Bluespec Inc (baseline RISC-V processors) and the computer architecture group at the University of Cambridge to extend the baseline processor to support CHERI. That is, it is ***not*** a novel contribution of my own for this thesis. However, I simply synthesise different variants of the processor to evaluate the area and static power consumptions for comparisons and report the results for completeness.

5.4 Development Contributions of My Own

On the software side, I use a baseline open-source FreeRTOS to build on. All of the development efforts on top of that are contributions of my own, including:

1. Porting FreeRTOS to use CHERI in purecap mode.
2. Porting RTEMS to use CHERI in purecap mode.
3. Extending RISC-V's Spike simulator to support CHERI ISA.
4. Porting FreeRTOS to run on Spike and QEMU.
5. Adding support to FreeRTOS for RISC-V's PMP/MPU.
6. Porting libdl from RTEMS to FreeRTOS and extending it to support CHERI and CompartOS.
7. Adding support for CompartOS and software compartmentalisation to FreeRTOS.
8. Implementing fault handling techniques.
9. Adding GPREL addressing support to CHERI-LLVM/Clang toolchain.
10. Porting VirtIO library to FreeRTOS and developing a network driver for it.
11. Porting all of the benchmarks and use cases to run on my FreeRTOS variants.

Hardware Platforms	Extending Spike simulator to support CHERI.
Toolchain	Adding GPREL addressing for FreeRTOS compartmentalisation in CHERI-LLVM.
	Adding GDB support to debug dynamically loaded compartments in FreeRTOS.
RTOSes	Porting FreeRTOS and RTEMS to CHERI and new hardware platforms.
CompartOS	Designing and implementing a generic CHERI-aware secure loader (based on libdl).
Applications	Porting benchmarks and applications to run on FreeRTOS.

Table 5.1: Engineering and development contributions

Target Frequency (MHz)	100.0
Pipeline Stages	5
RISC-V's Supported Features	RV32IMAC
RISC-V Privilege Modes	M, S, U
Caches	8KB L1 2-way set associative
TLB	16 entry direct-mapped

Table 5.2: Hardware configurations of the Bluespec's Flute processor used for evaluation.

I use a FreeRTOS BSP from Galois, Inc. as a baseline which includes network device drivers for the VCU-118 FPGA board I use. The ECU use case in Section 5.14 is also developed by Galois, Inc., and I deploy the CompartOS model to it to provide compartmentalisation and fault handling. Galois, Inc. has used my variant as one of their DARPA demos [122]. My contributions are summarised in Table 5.1. The work on simulation platforms (e.g., Spike and QEMU) is not directly reported here but has been useful for initial system bring-up and debugging. Similarly, a work-in-progress CHERI-RTEMS port exists but is not discussed in this evaluation. The port has been useful to get some experience on the porting effort to CHERI and generality and applicability of CompartOS to RTOSes other than FreeRTOS. That said, there are no plans to strongly evaluate or demonstrate CHERI-RTEMS for the purpose of this dissertation (though it can be a future work), and that is why I do not discuss it further in this dissertation.

5.5 Hardware Measurements

We use an FPGA softcore called Flute by Bluespec (see Section 2.10.5) which is a RISC-V processor and is comparable to Armv7 low-end cores (e.g., A9) that are commonly used in embedded systems (see Section 2.1). We synthesise baseline Flute with the hardware configurations in Table 5.2.

Flute is configurable when it comes to the hardware features and extensions it can include. RV32IMAC means that it is a 32-bit RISC-V core with a multiplication unit, atomics, and compressed instructions. The privilege modes are similar to Arm's privilege rings. For example, Machine-mode (EL3) is used for firmware, embedded systems or RTOSes, Supervisor-mode (EL1) runs conventional OSes that support paging and virtualisation such as Linux and FreeBSD, and User-mode is used for user applications. We build different variants of processors with different subsets of privilege modes to evaluate the area and power. Flute is part of an SoC that runs at 100 MHz on a VCU-118 FPGA board with networking, IIC sensors, and other peripherals. FPGAs, in general, are useful for hardware evaluation and rapid prototyping. They are cycles accurate but are generally slower in terms of frequency compared to ASIC processors. That said, FPGAs give us the ability to deeply analyse the hardware implications with accurate

	NOPROT	**PMP**	**MMU_PMP**	**CHERI**
RISC-V Privilege Modes	M	M, U	M, S, U	M
Protection	N/A	PMP (MPU)	PMP, MMU	CHERI
PMP regions	N/A	16	16	N/A
TLB	N/A	N/A	16 entry direct-mapped	NA

Table 5.3: Hardware configurations of different Bluespec processors we build for evaluation.

	NOPROT	**PMP**	**MMU_PMP**	**CHERI**
LUTs	92222	99810	101132	102955
Registers	119012	121498	122096	120600
Power (W)	0.212	0.213	0.221	0.245

Table 5.4: Power and area results for different variants of Flute

performance counters (e.g., cache misses, traps, capability-related operations) that we can not do with off-the-shelf ASIC processors (e.g., Arm). Further, there has been no CHERI-based ASIC processors that we can use for the purpose of evaluation during this period of research. That said, there are new CHERI-based Arm processors that have been released in 2022 [109]. However, they are targetting general-purpose operating systems such as FreeBSD, Linux, and Android; thus, it is not suitable for the scope of our embedded system. We hope the efforts in this dissertation and particularly the realism of evaluation will encourage industry to consider CHERI-based processors for embedded systems.

We use that setup for the remainder of this chapter as our evaluation platform. Flute is also extended to support CHERI-RISC-V [108]. We have built four different processors of Flute for evaluation, as shown in Table 5.3. The remaining configurations are the same as in Table 5.2.

NOPROT only has Machine-mode and does not have any protection units, which makes it the smallest processor configuration.

PMP is analogous to embedded M-class high-end processors that have two privilege modes for an embedded OS and user applications and an MPU unit. It has 16 regions (the maximum that the RISC-V specification allows), and it does not have MMU or TLBs that only exist in Supervisor-mode.

MMU_PMP corresponds to low-end, A-class processors that are able to run Linux, for example. It has both MMU and PMP units. Typically, feature-rich RTOSes run on such processors and optionally use PMPs for coarse-grained security.

CHERI solely relies on the CHERI-RISC-V ISA to enforce protection and compartmentalisation. It only has Machine-mode as it is believed to be sufficient for fine-grained, scalable security for embedded systems. This variant is the one CheriFreeRTOS uses. As we use 32-bit Flute, CHERI extends the 32-bit RISC-V integer registers to accommodate 64-bit compressed capabilities (see Section 2.10).

We have synthesised the four variants to evaluate the area consumption each takes shown in Table 5.4. The synthesising process and reported numbers are done by the Xilinx Vivado toolchain. We provide these numbers just for completeness with a notice that we mostly rule

	PMP	MMU_PMP	CHERI
LUTs	7.6%	8.8%	10.4%
Registers	2.0%	2.5%	1.3%
Power (W)	.47%	4.1%	13.5%

Table 5.5: Power and area percentage overheads of Flute variants compared to NOPROT.

out the area and power requirements as we target high-end embedded processors, discussed in Section 2.1. Still, the CHERI implementation could benefit from further hardware optimisations that have not been carefully sought.

Look-Up Table (LUT)s are the number of logic gates used in Xilinx FPGAs and represent the area consumption taken by the core logic. Registers are flip-flops used for fast storage. Finally, the power row in the table represents the approximate static power consumption, but that might not be a good indication of the runtime power consumption as that depends on the software workload and IO operations (e.g., DRAM and peripherals access).

As shown in Table 5.5, we find that the **PMP** variant with 16 regions adds 7.6% LUTs overhead compared to **NOPROT**, while the **MMU_PMP** one takes slightly more. **CHERI** takes the most LUTs overhead of 10.4% as it implements a rich capability-based ISA. On the other hand, **CHERI** takes the least Registers area overhead of 1.3% compared to 2% and 2.5% overheads for **PMP** and **MMU_PMP**, respectively, as **CHERI** does only run in Machine-mode and thus requires no further Supervisor/User mode registers, TLBs, or PMPs. We think that such overheads are low for the security benefits that CHERI and CompartOS bring.

5.6 Software Variants Security Implications

We have evaluated multiple FreeRTOS-based benchmarks and case studies on the previously discussed hardware variants and the following software setups:

- **INSECURE-STATIC:** Statically-linked, unmodified (and unprotected) baseline RISC-V software built without CHERI, PMP, or MMU support.
- **INSECURE-DYNAMIC:** Same as **INSECURE-STATIC**, but supports dynamically loading and linking RISC-V modules using `libdl`. This is running on the **NOPROT** hardware processor.
- **PMP-4-TASKS:** Statically-linked RISC-V software providing coarse-grained task-based and MPU-based compartmentalisation (4 regions). This is running on the **PMP** hardware variant.
- **PMP-N-OBJS:** Automatic linkage-based compartmentalisation where every object module (source file) is a compartment protecting N number of resources and protection is enforced using PMP/MPU, and is dynamically loaded and linked. This is running on the the **PMP** hardware processor.
- **PMP-N-LIBS:** Like **PMP-N-OBJ**, but compartments are library modules rather than object modules.

- **PURECAP:** Statically-linked pure-capability CHERI-RISC-V software providing complete (spatial) pointer safety, but no compartmentalisation, running on the the **CHERI** hardware processor.
- **COMPARTOS-OBJS:** Automatic linkage-based compartmentalisation where every source file (object module) is a compartment, protection is enforced using CHERI, and is dynamically loaded and linked. This is running on the **CHERI** hardware processor.
- **COMPARTOS-LIBS:** Like **COMPARTOS-OBJ**, but compartments are library modules rather than object modules.

We have implemented all of those software variants on top of FreeRTOS so that we get fair and comparable results against state-of-the-art systems and baselines on the same hardware.

PMP-4-TASKS is implemented to represent most embedded systems that provide optional coarse-grained protection mainly to protect the kernel from applications, and a very restricted form of isolation between applications themselves, which are represented as tasks. We refer to this system as MPU-based/task-based compartmentalisation. Most off-the-shelf deployed RTOSes optionally provide that. Examples are FreeRTOS-MPU [23], RTEMS, TockOS, and Mbed/uVisor. Further research systems such as TrustLite [27] and TyTAN [28] support use cases using an enhanced MPU/PMP unit with more regions.

PMP-N-OBJS is implemented to compare against the most advanced and related MPU-based compartmentalisation to our work, presented in ACES [26]. It provides automatic file-based (or IO based) protection. N represents the number of MPU regions and resources a compartment protects. We applied two variants of N; the first is the most that the actual PMP hardware can accommodate (16 in RISC-V), and the second is varied depending on each compartment, which is emulating an unlimited number of PMP regions to evaluate the scalability of MPUs in general when the processor vendors keep trying to increase them as a half-way solution. This is discussed further below.

PMP-N-LIBS is the most advanced MPU-based linkage-based implementation, not implemented by any system other than us, but we implemented it to compare real-world systems compartmentalised around libraries, as we think file-based or object-based systems are not versatile or intuitive enough to be used in real embedded systems, giving the developer the option to create a compartment with multiple files and objects that form a library.

PURECAP represents a baseline CHERI-based system without automatic linkage-based compartmentalisation. It is provided to compare against (as previous work) and to get an idea about the overhead of the increased pointer sizes and CHERI's pure-capability generated code.

COMPARTOS-OBJS and **COMPARTOS-LIBS** are our main novel contributed systems that add automatic linkage-based compartmentalisation to **PURECAP**.

Table 6.1 summarises the security properties of the software variants.

[H]

Table 5.6: Security properties of the software variants.

Software Variant	Security Properties	Security Limitations	Protection Domain	Resources	Domain Switches	Hardware
INSECURE-STATIC	No security	No security	None	N/A	NA	NOPROT
INSECURE-DYNAMIC	No security	No security	None	N/A	NA	NOPROT
PMP-4-TASK	Task-based coarse-grained isolation. Each task can define and isolate 4 memory regions besides its stack. Coarse-grained separation between tasks and the kernel; system calls are used to perform privileged operations.	PMP hardware regions count. Tasks cannot define or isolate global variables or dynamically allocate memory. Manual compartmentalisation. No intra-task compartmentalisation.	Task/Process	Memory regions, IO, kernel and applications.	Task switches (IPC), system calls, reconfigure PMP.	PMP
PMP-16-OBJS	Linkage-based isolation, protecting source-code files and objects. Automatic compartmentalisation. Intra-task compartmentalisation.	PMP hardware regions count. Cannot represent libraries.	Source/object files.	Memory regions, IO, kernel and applications.	Inter-compartment function calls, system calls, reconfigure PMP.	PMP
PMP-N-OBJS	Same as PMP-16-OBJS, but emulates enforcing pointer safety using PMP/MPU.	Cannot represent multiple source-files/libraries.	Source/object files.	Pointers, memory regions, IO, kernel and applications.	Inter-compartment function calls, system calls, reconfigure PMP.	PMP
PURECAP	Capability-based security, protecting every pointer.	No compartmentalisation support to define a unit of protection, and thus no availability or recovery support when a fault or violations occur.	Task/Process	Pointers, memory regions, IO, kernel and applications.	Task switches (IPC). Reloading capability register file per task.	CHERI
COMPARTOS-OBJS	Capability-based security, protecting every pointer. Linkage-based isolation, protecting source-code files and objects. Automatic compartmentalisation. Intra-task compartmentalisation.	Cannot represent source-files libraries.	Source/object files and tasks/processes.	Pointers, memory regions, IO, kernel and applications.	Task switches (IPC). Inter-compartment function calls, reloading a single root capability table register.	CHERI
COMPARTOS-LIBS	Same as COMPARTOS-OBJS	None	Source/library files and tasks/processes.	Pointers, memory regions, IO, kernel and applications.	Task switches (IPC). Inter-compartment function calls, reloading a single root capability table register.	CHERI

5.7 MPU/PMP-based Compartmentalisation Emulation

When applicable, we discuss two variants with 16 PMP regions (hardware restriction) and an emulated PMP implementation without limitations to the region count to provide a fair comparison against fine-grained CompartOS variants by protecting the same number of resources per compartment. Otherwise, it is unfair (from a security and performance perspective) to compare CompartOS which protects every pointer, object file, library, and every compartment (those can scale to hundreds), against a system that is only protecting 16 regions of memory or less. The emulation basically performs all that a PMP-based implementation would do, including maintaining an ACL to describe MPU regions per compartment, reloading them from memory, performing system calls to finally reconfigure the PMP hardware. Reconfiguring the PMP is an O(N) operation where N is the number of resources and regions to protect. Providing this emulated PMP variant gives an idea about how scalable PMP/MPU are from a software perspective, as the current embedded processor architectures tend to increase them as a halfway solution for scalability. For example, Armv8 and TrustLite increased the number of MPU regions to 32 instead of 8 or 16, which was the usual case in Armv7 processors.

5.8 Limitations

In this section, we list the limitations and things we have not considered doing and propose potential solutions and future work. Those limitations fall within our threat model discussed in Section 3.1. First, we only consider a specific scope of embedded systems that does not have restrictions on memory size and footprints. That is, we do not consider tiny, low-power embedded systems. While our work could fit there in as small as 100 KiB of memory, we think such systems do not have scalability or fine-grained security as major requirements compared to the memory footprints and power consumption, especially with a few compartments and resources. Further, the secure dynamic linker and loader is memory and CPU bound (at boot time). Thus, we have omitted evaluating memory size and power consumption. We also do not try to compare against MMU-based protection as they are subtly different, and in most EOSes, they are not used as they do not meet the real-time, determinism, and single-address-space design requirements. Further, we do not try to evaluate the confidentiality or integrity but rather focus on the performance, compatibility, scalability, and recovery of the compartmentalisation model "after" catching security violations. CHERI, by design, catches most memory-safety vulnerabilities, and there have been already a sound evaluation in CHERI literature about its security impact. Further, CompartOS not only catches CHERI violations but also helps with other types of faults as well (e.g., asserts). That said, future work of measuring the isolation and protection degree of each compartment can be applied. For instance, we could have dynamic tracing that composes a tree or graph of all capabilities a compartment has access to at a specific runtime point and compare that against an expected static policy. Formal verification of the secure dynamic loader could also be an option given that it is relatively small and is the only software TCB component. This is in contrast to Rust, where the toolchain is also part of the TCB, besides the runtime software such as TockOS. We do not further implement or evaluate temporal safety or revocation (see Section 4.5) as they are not part of our requirements or threat model. That said, the model allows for revocation strategies, and we have prototyped with simple techniques there that could be part of future work. On the security policy, we only evaluate the default semantics of symbols visibility and scopes as dictated by a programming language and ELF linkage model. For instance, a linkage-based library compartment can have static variables and functions that cannot be dereferenced by other compartments. Further, the visibility of all global symbols in one compartment is hidden by default, except for the API (as specified by the developer/designer), which can be minted or dereferenced by other compartments. We do not try to evaluate further custom policies around IO, abstract software resources, or restrict interactions between compartments beyond the API. Such custom policies are very application-specific, and hence separate use cases (e.g., workshop papers) with very well-defined security policies, including compartments and their restricted interaction, could be regarded as future work. Thus, we only provide a mechanism rather than a policy. Finally, we do not evaluate execute-in-place as this is mostly a requirement for tiny embedded systems, and we also do not support shared object linking and loading yet that is required to avoid fixing up relocations in the text code segment itself that usually exists in the ROM. However, this could be regarded as future work as well.

5.9 Compatibility and Disruptiveness

In this section, we analyse the amount of effort required to support protection and compartmentalisation on the OS and application level. We demonstrate what it takes to support **PMP-4-TASKS**, **PURECAP**, and **COMPARTOS** models by applying them to our insecure baseline FreeRTOS prototype and application use cases. This is a good speculative indicator to evaluate compatibility and disruptiveness for insecure embedded systems while adopting one of these models to entertain some form of security and protection.

5.9.1 Securing Embedded Operating Systems

Based on my experiences porting embedded operating systems (FreeRTOS and RTEMS) to support MPU/PMP, CHERI-RISC-V, and further compartmentalisation support, I categorise the changes and disruptiveness of porting an embedded OS to one of the security models as follows:

1. **PMP-4-TASKS**: to enforce task-based compartmentalisation using PMP/MPU.
2. **PURECAP**: to enforce complete pointer safety using CHERI.
3. **COMPARTOS**: to enforce linkage-based compartmentalisation and fault handling.

Figure 5.1 shows the number of LoC changes (insertions and deletion) that were required in order to support each protection model, porting FreeRTOS to use RISC-V's MPU/PMP, CHERI-RISC-V, and supporting CompartOS.

PMP-4-TASKS requires changes to set up MPU regions during boot time in order to protect the kernel from user applications, effectively providing coarse-grained security between two types of protection domains: the kernel and applications. The context switch code, written in assembly, needed to support reconfiguring RISC-V's PMP base, bounds, and configuration registers for the user-defined memory regions per task, including its stack. Part of the task/threading subsystem also needs to add bookkeeping for each task that holds an ACL of memory regions and be able to store and update this ACL on task creation and further parametrisation or updates. Finally, a significant target-independent code of FreeRTOS needed to be implemented to support system calls and privilege escalation, effectively wrapping every FreeRTOS API with an MPU wrapper. It is worth noting

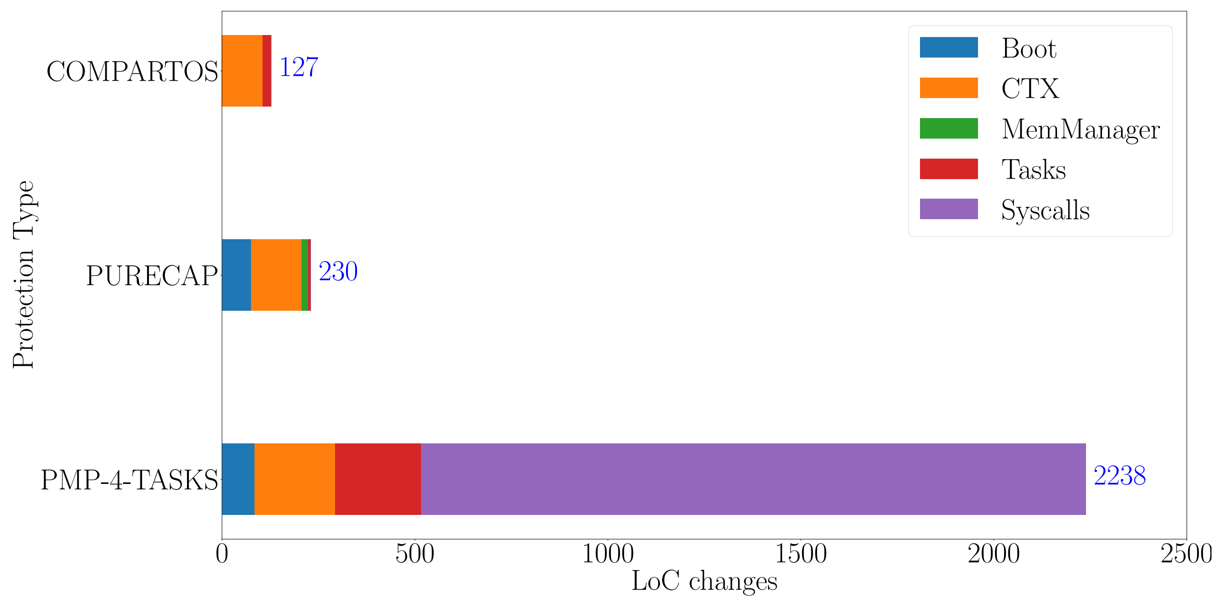


Figure 5.1: Lines of code changes to implement and support protection models in FreeRTOS. CTX is context switching and low-level ISR handling in assembly code. MemManager is the heap allocator in FreeRTOS. Tasks include added code during creation, bookkeeping and state updates. Syscalls only exist in MPU/PMP, where there are two privilege modes for the unprivileged application and the kernel.

that FreeRTOS-MPU does not try to provide memory protection or isolation during heap allocations. Rather, it prevents unprivileged tasks from calling malloc/free. This is to avoid the complexity and hardware restrictions of MPUs/PMPs (as discussed) doing so.

PURECAP is enabling complete pointer safety using CHERI. It is straightforward to support PURECAP in an existing embedded OS with minimal source-code changes. Early boot code that is written in assembly and C needs to limit root-capability privileges, initialise capabilities for all pointers, and create restricted ones, then hand them over to other subsystems. The context-switch and ISR handlers that are written in assembly need to be rewritten with capability registers instead of integer registers. Unlike **PMP-4-TASKS**, **PURECAP** requires the memory allocator to create bounded capabilities on all returned memory regions (e.g., called by malloc). This only required 17 LoC additions to the FreeRTOS memory manager.

COMPARTOS requires very minimal changes to the OS itself, as discussed in Chapter 4. The changes are to bookkeep and attach a compartment context per task, similar to **PMP-4-TASKS**. The context switch code also needs to check if, when handling a fault, it is executing in a compartment, and then it jumps to a compartment-specific fault handler before returning to the caller compartment. The remaining trusted COMPARTOS component is the secure loader that includes dynamic loading, policy checks, generating trampolines, creating capabilities per compartment, and performing fault handling. The secure loader is OS independent and can be used with other embedded OSes as well; thus, it is not affecting an existing OS' compatibility or disrupts its implementation like **PMP-4-TASKS**.

5.9.2 Securing Applications

Enforcing pointer safety (i.e., PURECAP) at the application level requires no code modifications for cleanly-written C code that does not abuse pointer arithmetic. It would just require rebuilding the application with the proper compiler flags. This makes it possible for a large codebase that is already written in C to immediately entertain full pointer safety without too much effort. For example, most existing FreeRTOS applications and demonstrations run by simply being rebuilt. We were able to rebuild and use TCP/IP, HTTP, FTP, SSL/TLS, MQTT, OTA, and FAT libraries and applications in Sections 5.11, 5.12, 5.13, 5.14, without any source-code changes. On the other hand, trying to protect applications using task-based isolation (i.e., PMP-4-TASKS) requires non-trivial efforts even for the smallest use cases, as discussed in Section 5.10.2.

Supporting compartmentalisation (i.e., CompartOS) from the application perspective requires some extra efforts to:

- Define each logical compartment in the system, and tell the build system about it.
- Define a security policy that defines how different compartments in the system may interact.
- Optionally provide a custom fault handler for each compartment in the system.

5.10 Microbenchmark Evaluation

We have developed this custom microbenchmark from scratch to measure the performance of the lowest level, most critical and frequently executed code paths in embedded OSes in the different software variants that we discussed. This is to evaluate the overheads compared to unprotected baseline, PMP variants, CHERI, and CompartOS.

Figure 5.2 represents a linkage-based compartmentalised system of the microbenchmark. There are two main communicating compartments: a *sender* and a *receiver*. Communication is done in different ways, such as normal function calls, compartment calls, and IPC message-passing. Listings 5.1 and 5.2 show the pseudocode of both compartments.

In the object-based compartmentalised variants (e.g., COMPARTOS-OBJS and PMP-N-OBJS), each compartment is an object file. The sender compartment has 18 resources to protect, and the receiver compartment has 41. Those are memory regions representing pointers, global variables, UART regions, and functions. The number of resources is automatically determined at build time depending on how many global symbols exist in the linkage module, which is an object file in this case.

In Section 5.10, we list the absolute numbers of the most critical and shortest execution paths. Generally speaking, in microbenchmarking and small numbers, the overheads of such paths are maximised and might not be representative of the overall performance of real-world workloads. However, they are vital to understanding particular aspects of protection domain switching and communication in the system. In section 5.10.3, we evaluate different domain switching mechanisms and their overheads. In section 5.10.4.1, we amortise IPC buffer sizes to evaluate performance with increased IPC and memory workloads, which slightly represents more realistic memory and communication-intensive workloads.

5.10.1 Runtime Performance Evaluation of Critical Operations

In this build, we measure major critical code paths between different software variants as follows:

```
void Func(void* arg) {
    EndTime();
    LogTimeDiff();
}

// Measure fault time (context switch and exception handling)
StartTime();
Fault();
EndTime();
LogTimeDiff();

// Measure a function call time
StartTime();
Func();

// Measure a compartment call time
StartTime();
ExternCall();
EndTime();
LogTimeDiff();

// Measure IPC time of sending one byte
StartTime();
FreeRTOS_xQueueSend(QueueHandle, &buffer, 1);

// Measure IPC time sending a fixed amount of 4096 bytes
// using varying buffer sizes
#define TOTAL_SIZE 4096
char buffer[TOTAL_SIZE ];

for (int e= 0; e <= 12; e++) {
    int buffer_size = pow(2, e);
    int queue_send_count = TOTAL_SIZE / buffer_size;

    StartTime();
    for (int i = 0; i <  queue_send_count; i++) {
        FreeRTOS_xQueueSend(QueueHandle, &buffer, buffer_size);
    }
}
```

Listing 5.1: Pseudocode of the sender compartment

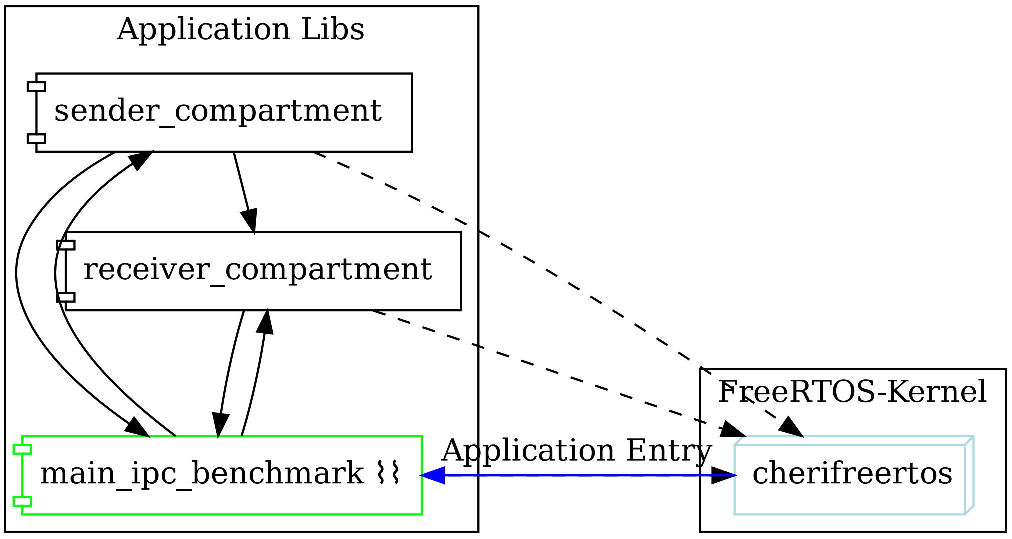


Figure 5.2: Microbenchmark compartments graph that is statically and automatically generated by CheriFreeRTOS during the build stage. Small boxes represent linkage-based object compartments, and the edges are the API references a compartment makes to another. Dotted edges indicate a trusted call to FreeRTOS without trampolines or domain switches. Compartments are placed into larger boxes to categorise them into core, third-party, and application libraries.

CONTEXT_SWITCH A yield system call to measure the context-switch assembly code latency.

COMP_FAULT The overhead of a compartment fault until it returns.

FUNC_CALL A normal C function call **without a trampoline**.

COMP_CALL A call to an external global function in another compartment that triggers a **compartment switch** via a **trampoline**.

TASK_NOTIFY FreeRTOS inter-task communication using notifications.

TASK_QUEUE FreeRTOS inter-task communication using queues.

COMP_FAULT and **COMP_CALL** only exist in compartmentalised systems that support fault handling and recovery and so cannot be measured for non-compartmentalised configurations. **COMP_CALL** represents linkage-based domain switching, while **TASK_QUEUE** is effectively a task-based domain switching using IPCs sending one byte from one task to another. While sending one byte over IPC is common in embedded systems (e.g., sending a code command to start, stop, or resume a hardware device), the overhead of such operation is maximised. Later on, in Section 5.10.4.1, we increase the IPC buffer sizes to amortise overheads to analyse the impact on communication and memory-intensive workload.

Table 5.7 reports the number of instructions and cycles results on the Flute variants. First, we evaluate the hypothesis that dynamic linking and loading does not add substantive extra

```
void ExternCall(void* arg) {
    EndTime();
    LogTimeDiff();
}

StartTime();
FreeRTOS_xQueueReceive(QueueHandle, &buffer, 1);
EndTime();
LogTimeDiff();

// Measure IPC time after receiving data
#define TOTAL_SIZE 4096
for (int e= 0; e <= 12; e++) {
    int buffer_size = pow(2, e);
    int queue_send_count = TOTAL_SIZE / buffer_size;

    for (int i = 0; i <  queue_send_count; i++) {
        FreeRTOS_xQueueReceive(QueueHandle, &buffer, buffer_size);
    }
    EndTime();
    LogTimeDiff();
}
```

Listing 5.2: Pseudocode of the receiver compartment

	CONTEXT_SWITCH		COMP_FAULT		FUNC_CALL		COMP_CALL		TASK_NOTIFY		TASK_QUEUE	
	Instructions	Cycles	Instructions	Cycles	Instructions	Cycles	Instructions	Cycles	Instructions	Cycles	Instructions	Cycles
INSECURE-STATIC	177	475	0	0	8	118	9	116	495	2087	668	2433
INSECURE-DYNAMIC	177	475	0	0	8	118	9	72	495	2212	668	2620
PMP-4-TASKS	204	552	0	0	8	118	9	114	697	2585	867	3198
PMP-16-OBJS	391	1246	409	1193	8	116	240	754	1053	3784	1223	4550
PMP-N-OBJS	403	1270	589	1533	8	116	420	1151	1245	4327	1415	5006
PMP-N-LIBS	368	917	0	0	8	116	9	115	1007	3580	1177	4136
PURECAP	197	563	0	0	8	118	9	73	530	2149	753	2635
COMPARTOS-LIBS	197	745	0	0	7	118	8	73	532	2604	759	3539
COMPARTOS-OBJS	197	1002	230	1028	7	77	44	212	532	2778	759	3712

Table 5.7: Number of instructions and cycles for microbenchmarks

overhead by comparing **INSECURE-STATIC** and **INSECURE-DYNAMIC**. It is concluded to be true since both variants have the same number of instructions and similar cycles latencies (cycles slightly vary due to cache misses and branch mispredictions). As mentioned earlier, all the linkage-based compartmentalised variants (those that have OBJS or LIBS in their name), including PMP and COMPARTOS, are dynamically loaded and linked. The remaining variants are statically linked.

5.10.2 MPU/PMP Adaptation Cost

Comparing **PMP-4-TASKS** against **INSECURE-STATIC**, we had to refactor the existing benchmarking application code to use off-the-shelf FreeRTOS' MPU API [23] to create unprivileged tasks, MPU regions, map MPU regions to program resources (e.g., global variables, stack, and a UART region). The code redesign and refactor took 160 LoC changes (insertions and deletions) out of an overall of 387 LoC. That is, 41% of the application code had to be changed for such a small and simple application, even though it was originally designed to be task-based. With large, sophisticated applications with ∼100 KLoC (like the OTA demo discussed in Section 5.13), the process of refactoring the code to use MPU-based tasks is likely to be infeasible and impractical. Even if it is doable, it is going to take a considerable amount of effort. For example, in the OTA use case, third-party libraries such as mbedTLS (60 KLoC), tinycbor (1.5 KLoC), and OTA (4 KLoC) will all have to be redesigned to only execute around tasks, and communications between them will need to be implemented as IPC rather than simple function calls, which requires careful and manual modifications. Furthermore, each compartment will need to define a fixed set of resources (e.g., memory regions, IO, pointers) and perform system calls to attach resources to tasks and enforce permissions and isolation on them. This is compared to **PURECAP** and **CompartOS** variants that required no source-code changes at all, even for the most complex application like OTA. Based on this evaluation, we conclude:

Conclusion: In our FreeRTOS prototype, CompartOS is more compatible than manual task-based compartmentalisation models, prototyped in FreeRTOS with a minimalist application and similar to other deployed systems (like in TockOS, FreeRTOS-MPU, and uVisor/Mbed).

5.10.3 Performance Evaluation of Domain Switching Mechanisms

We compare CompartOS' linkage-based domain switch against state-of-the-art PMP/MPU task-based domain switch in Figures 5.3 and 5.4.

As mentioned, task-based domain switching is mapped to the **TASK_QUEUE** column in Table 5.7. In the **PMP-N-OBJS** variant, each compartment is attached to a task; thus, **PMP-N-OBJS/TASK_QUEUE** represent Task-based compartmentalisation using PMP/MPU. Looking at Figure 5.3, we notice that the instructions are consumed in performing system calls, IPC/kernel path, and PMP configurations. On the other hand, CompartOS' task-based compartmentalisation (see Section 3.2) does not require any system calls or PMP reloads, and compartment switches happen implicitly during task context switches when the CGP register is swapped. Task-based-CompartOS in Figure 5.3 is mapped to **COMPARTOS-OBJS/TASK_QUEUE** in Table 5.7. Based on that, we conclude that:

Conclusion: In our FreeRTOS prototype, CompartOS' task-based domain crossing is 26.6% faster than MPU-based, task-based, IPC mode, implemented by state-of-the-art deployed systems (like in FreeRTOS-MPU and TockOS).

Similarly, we compare CompartOS linkage-based domain switch against the state-of-the-art MPU-based linkage-based systems (e.g., ACES, uVisor, TrustLite/TyTan) in Figures 5.3 and 5.4. Those are mapped to the **COMP_CALL** column in Table 5.7. As shown, the PMP reconfigurations and system calls dominate the cost of Linkage-based-PMP; both are not required in CompartOS.

Conclusion: In our FreeRTOS prototype, CompartOS' linkage-based domain switch is 85% faster than the most similar MPU-based compartmentalisation state-of-the-art systems.

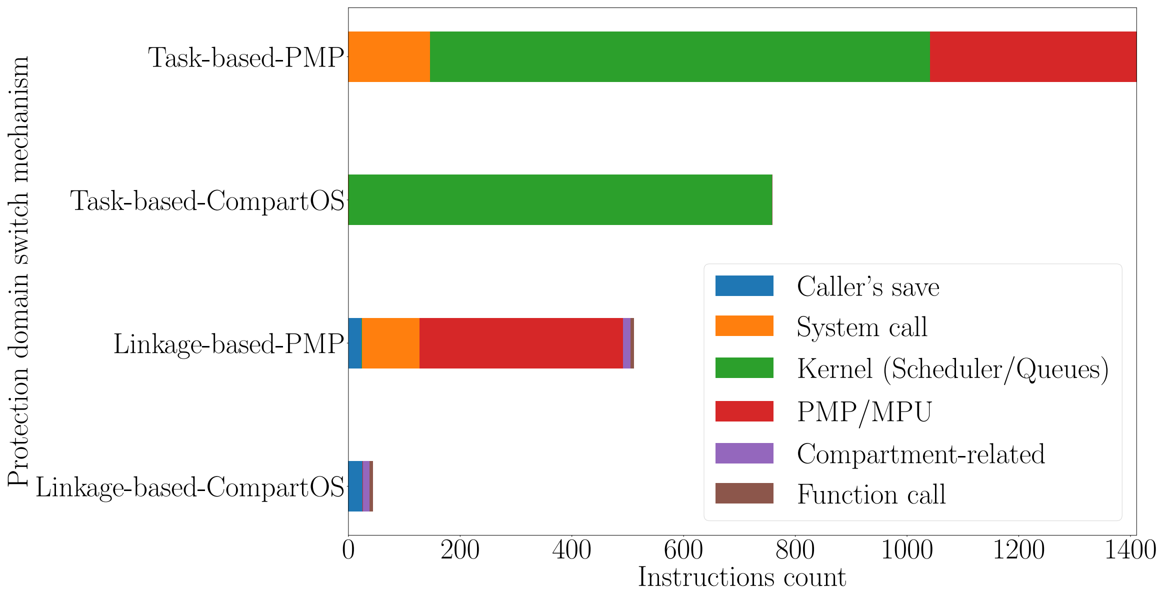


Figure 5.3: Comparison between different protection domain switch mechanisms cost in instructions count.

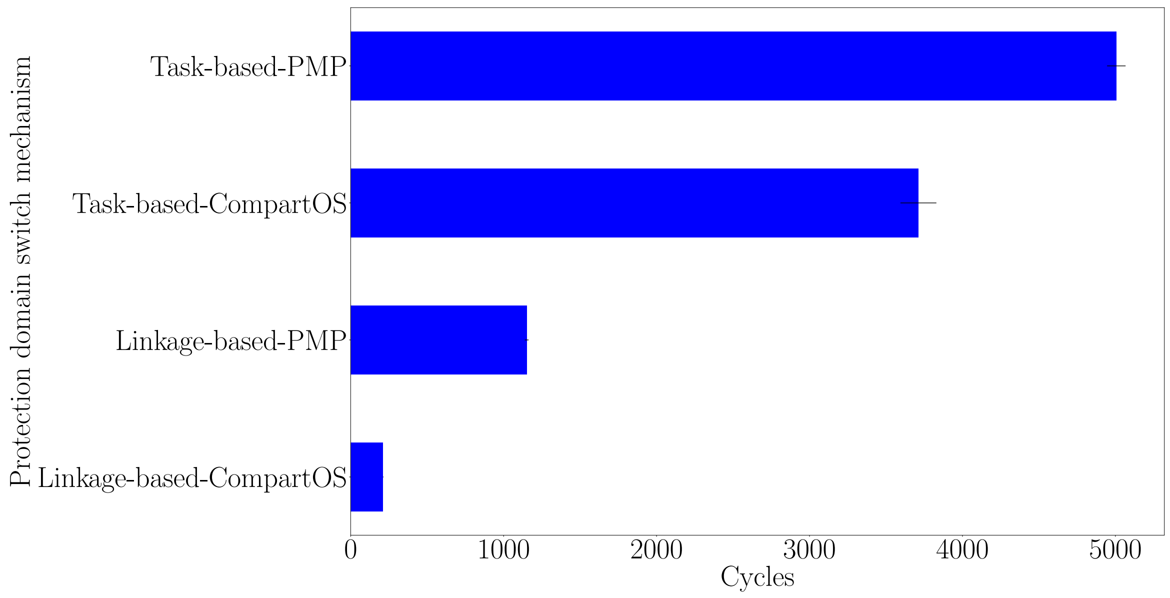


Figure 5.4: Comparison between different protection domain switch mechanisms cost in cycles count.

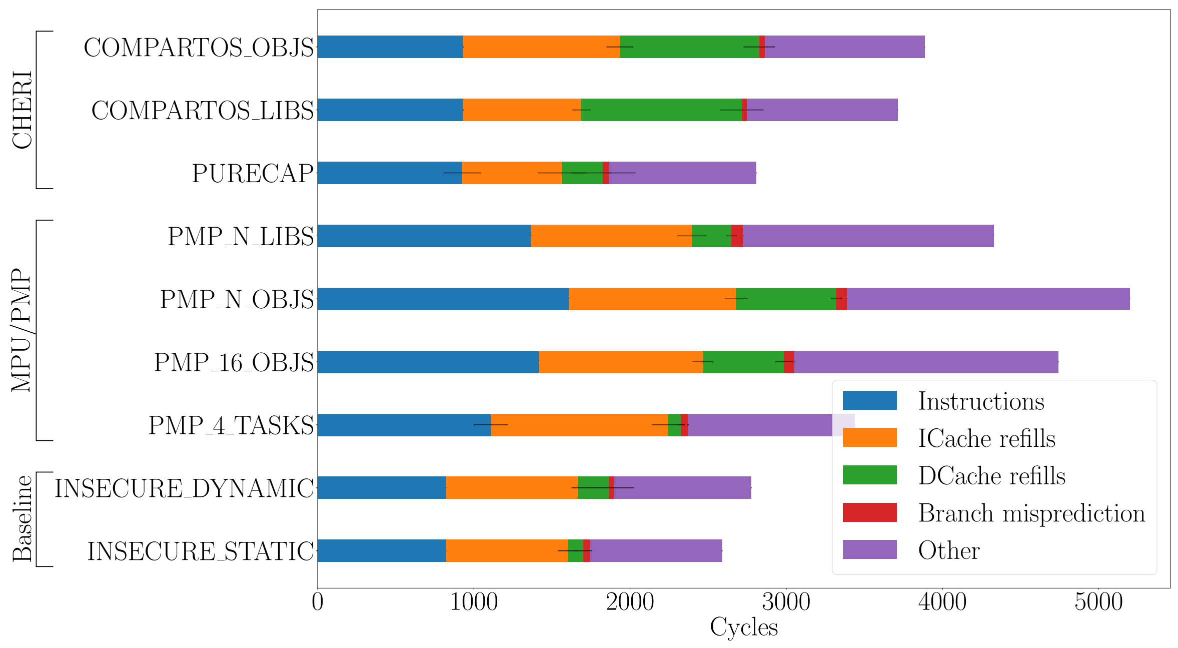


Figure 5.5: The cost, in cycles, of task-based domain switching (TASK_QUEUE) sending one byte from one task to another.

Finally, we compare CompartOS linkage-based compartmentalisation against off-the-shelf deployed OSes that rely on task-based, MPU-based compartmentalisation. CompartOS offers intra-thread domain switches thanks to its linkage-based (rather than task-based) design. Comparing **COMPARTOS/COMP_CALL** against **PMP-N-OBJS/TASK_QUEUE**, we conclude that:

Conclusion: In our FreeRTOS prototype, CompartOS' linkage-based domain switch is 95% faster than the off-the-shelf message-passing IPC in task-based, MPU-based models implemented by state-of-the-art deployed systems (e.g., FreeRTOS-MPU and TockOS).

5.10.4 Task-based IPC Evaluation

In secure microkernels and RTOSes (like in seL4, FreeRTOS-MPU, and TockOS), secure domain switches are often task-based using IPCs. **TASK_QUEUE** resembles that. In Figure 5.5, we show the performance overhead of performing task-based domain switches, sending 1 byte over a queue. Each bar contains segments representing where cycles are consumed. Some instructions (e.g., accessing system registers, division, multiplication, trapping and returning, etc.) are multi-cycle and/or require pipeline flushes; those are placed in the *Other* segment. Other instructions take one cycle, and they are accounted for in the *Instructions* segment. The remaining segments are cache misses and branch misprediction latencies. The PMP variants take the most instructions represented in *Instructions* and *Other* segments as they have to perform system calls, reconfigure PMP registers that require loading permissions from memory and writing the PMP system registers. COMPARTOS variants do incur negligible overhead in data cache misses, likely because of the partitioned capability table per compartment, unlike PURECAP. Figure 5.6 shows the overheads of task-based domain switching compared to **INSECURE-STATIC**.

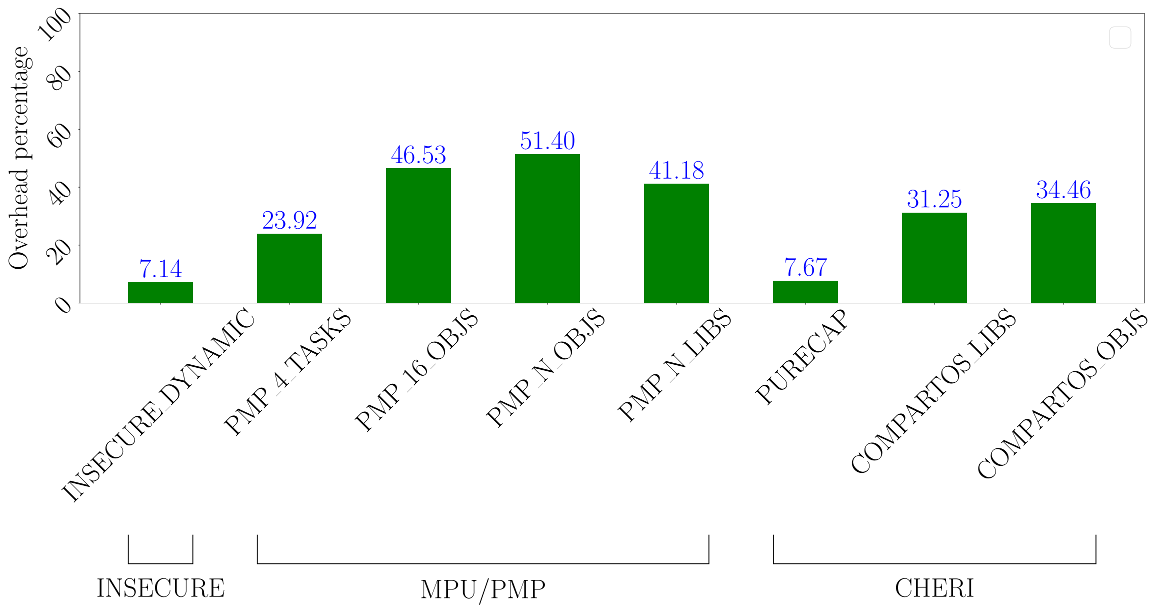


Figure 5.6: Task-based domain switching (TASK_QUEUE) overheads, sending one byte from one task to another and relative to INSECURE_STATIC baseline.

5.10.4.1 IPC Buffer Size Amortisation

We amortise the IPC performance overheads by scaling the IPC workload between compartments as follows:

1. Scale N in the **PMP-N-OBJS**.
2. Scale the IPC workload by scaling the number of send/receive operations between compartments.

Thus, in this section, we only consider linkage-based compartmentalisation enforced by both PMP and CompartOS/CHERI. This gives us a more realistic performance evaluation compared to microbenchmark that is only sending one byte during IPC; thus, the overhead is maximised there. First, N is scaled from 16 to 41 if we compare **PMP-16-OBJS** against **PMP-N-OBJS**. We notice 17% and 21% overheads in the **COMP_FAULT** and **COMP_CALL** cycles, respectively. This is to context-switch the PMP context and reconfigure the hardware. Both overheads will scale to O(N). This suggests that MPU-based compartmentalisation techniques are not scalable, performance-wise. CompartOS, on the other hand, does not incur O(N) overheads during either path as the only thing required during a **COMP_FAULT** or **COMP_CALL** is to change the root captable capability held in the CGP register.

Second, looking at Figure 5.7, we measure the cycles taken to send fixed-size data of 4 KiB using different buffer and queue sizes. This scales the number of send/receive calls, tasks context switches, and cache pressure. When the overall buffer size is 4 KiB, data is sent at once in a single call. In this case, **PMP-N-OBJS** only incurs 10.65% overhead compared to **INSECURE-STATIC**, while **CompartOS** is actually 32.49% faster because of the doubled register size of CHERI, which halves the number of copy instructions spent in *memcpy*. The cycles and instructions overheads are shown in Figures 5.9 and 5.10, respectively.

When we lower the buffer size to 2 bytes, this requires 2048 send/receive operations. The **PMP-N-OBJS** incurs a 70.29% overhead, while **CompartOS** incurs 41% mainly due to the

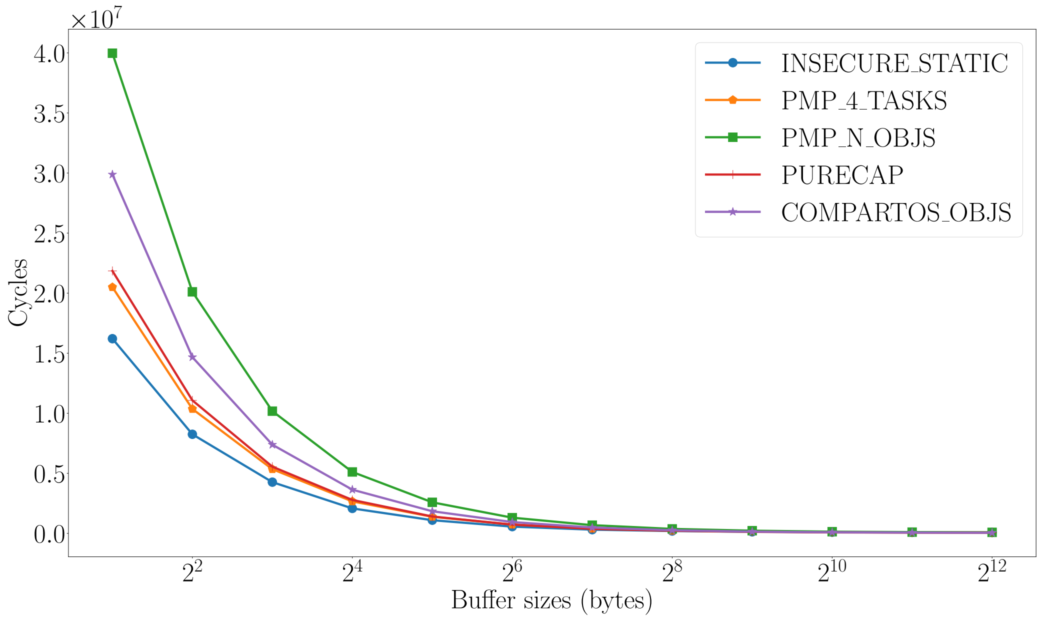


Figure 5.7: Performance, in cycles, when varying buffer sizes to send a fixed-size data of 4 KiB.

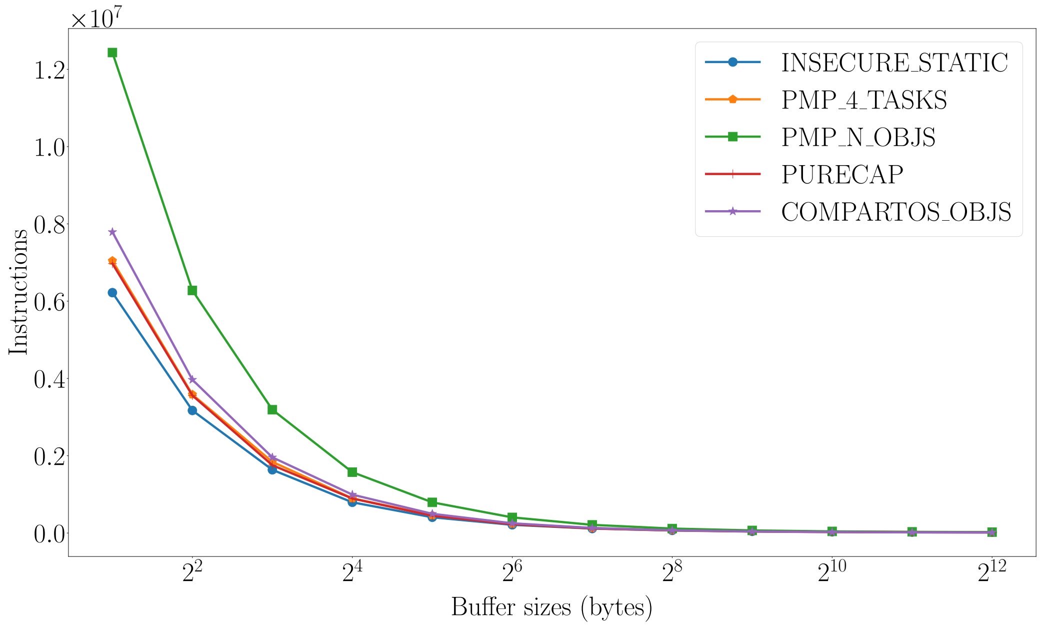


Figure 5.8: Performance, in instructions count, when varying buffer sizes to send a fixed-size data of 4 KiB. A CHERI-aware memcpy is used.

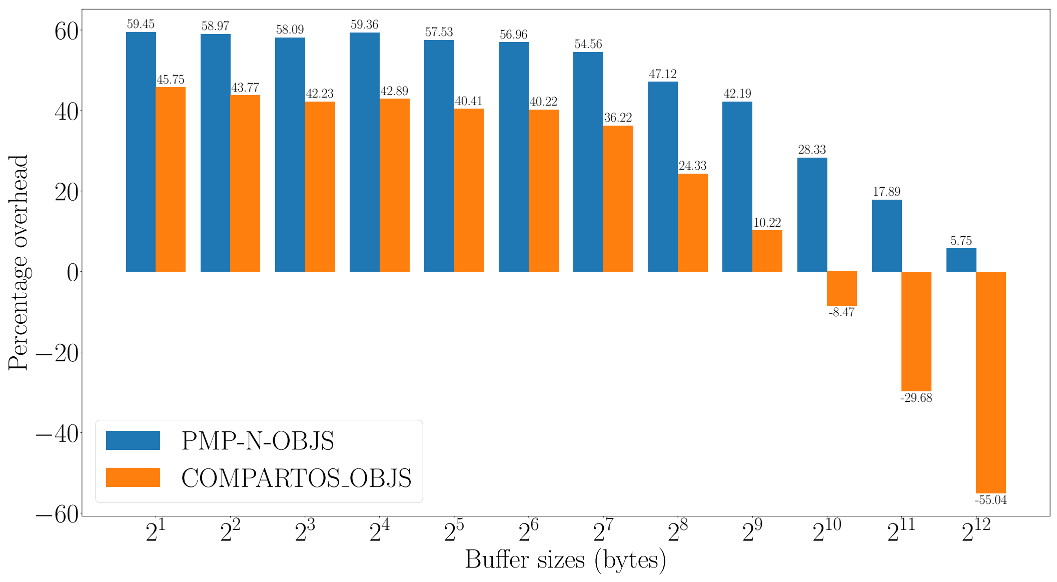


Figure 5.9: Overhead percentage of cycles compared to INSECURE-STATIC baseline when varying buffer sizes to send a fixed-size data of 4 KiB.

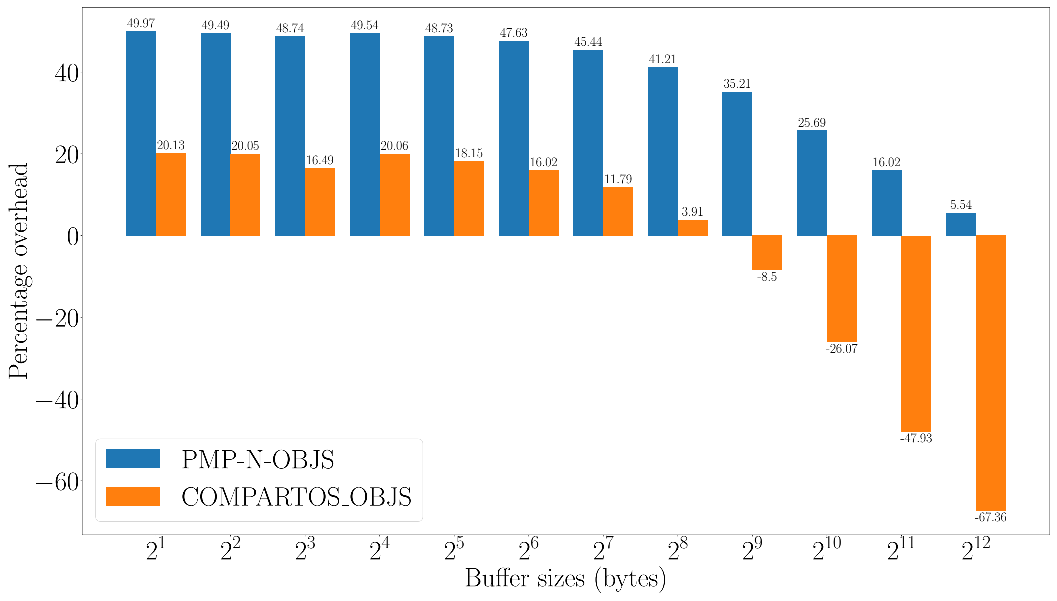


Figure 5.10: Overhead percentage of instructions count compared to INSECURE-STATIC baseline when varying buffer sizes to send a fixed-size data of 4 KiB. A CHERI-aware memcpy is used.

cache pressure of double-sized register spills and context switches. Furthermore, the *memcpy* advantage of **CompartOS** is gone when the buffer size is 4 bytes or less as the integer register is 4 bytes in 32-bit RISC-V and CHERI register is 8 bytes, which makes the number of required copy instructions the same, using a single register in both cases.

Even though the CHERI-aware memcpy boost is inherent and should not be disabled for any CHERI-based software, we have rerun the benchmarks without any memcpy advantage at all for CHERI and CompartOS by providing a target-independent memcpy. This strips the benchmark away from any advantage for CHERI and focuses on the overheads due to CHERI's cache pressure and compartmentalisation code such as trampoline switches and GPREL addressing. Figure 5.14 shows that CompartOS always incurs instructions overheads compared to baseline, without the memcpy advantage. Still, the overhead is less than half of its analogues **PMP-N-OBJS** variant. The cache pressure of CHERI's doubled pointer sizes, GPREL addressing, and trampolines are thus maximised, as shown in Figure 5.13. Still, **COMPARTOS-OBJS** scales well and outperforms **PMP-N-OBJS** when there are many trampoline switches (i.e., smaller buffer sizes), as shown in Figures 5.11 and 5.12.

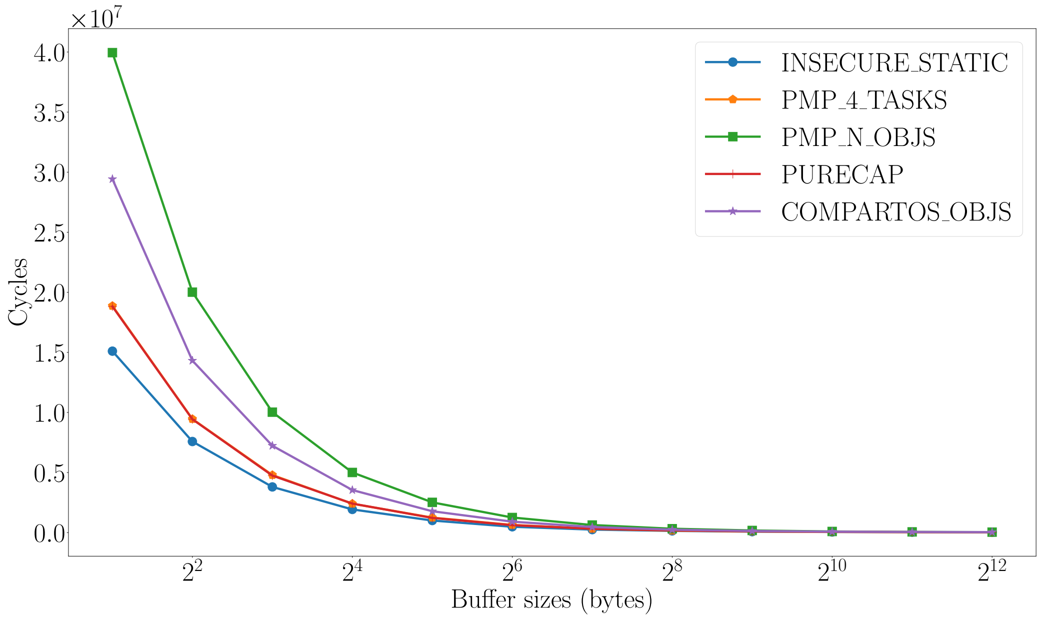


Figure 5.11: Performance, in cycles, when varying buffer sizes to send a fixed-size data of 4 KiB. A non-CHERI-aware memcpy is used, doing byte-by-byte copies across all software variants.

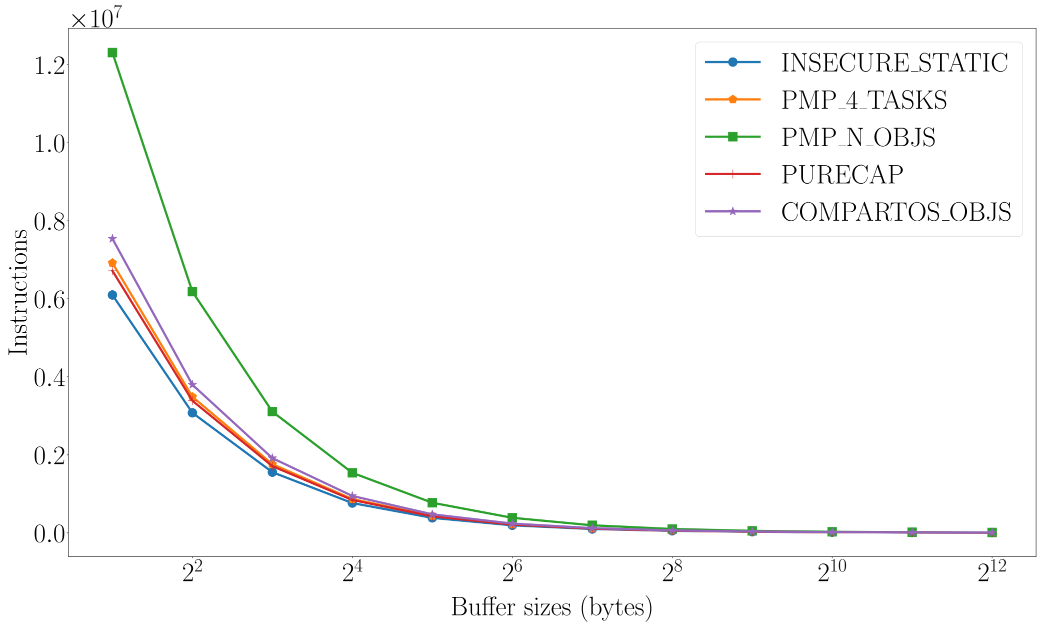


Figure 5.12: Performance, in instructions count, when varying buffer sizes to send a fixed-size data of 4 KiB. A non-CHERI-aware memcpy is used, doing byte-by-byte copies across all software variants.

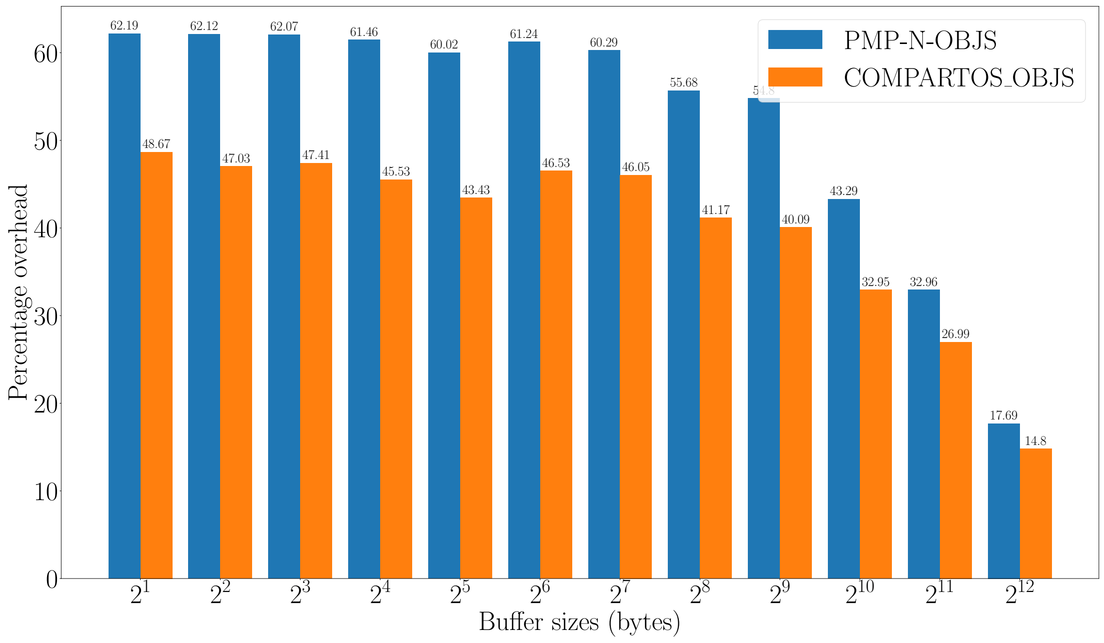


Figure 5.13: Overhead percentage of cycles compared to INSECURE-STATIC baseline when varying buffer sizes to send a fixed-size data of 4 KiB. A non-CHERI-aware memcpy is used, doing byte-by-byte copies across all software variants.

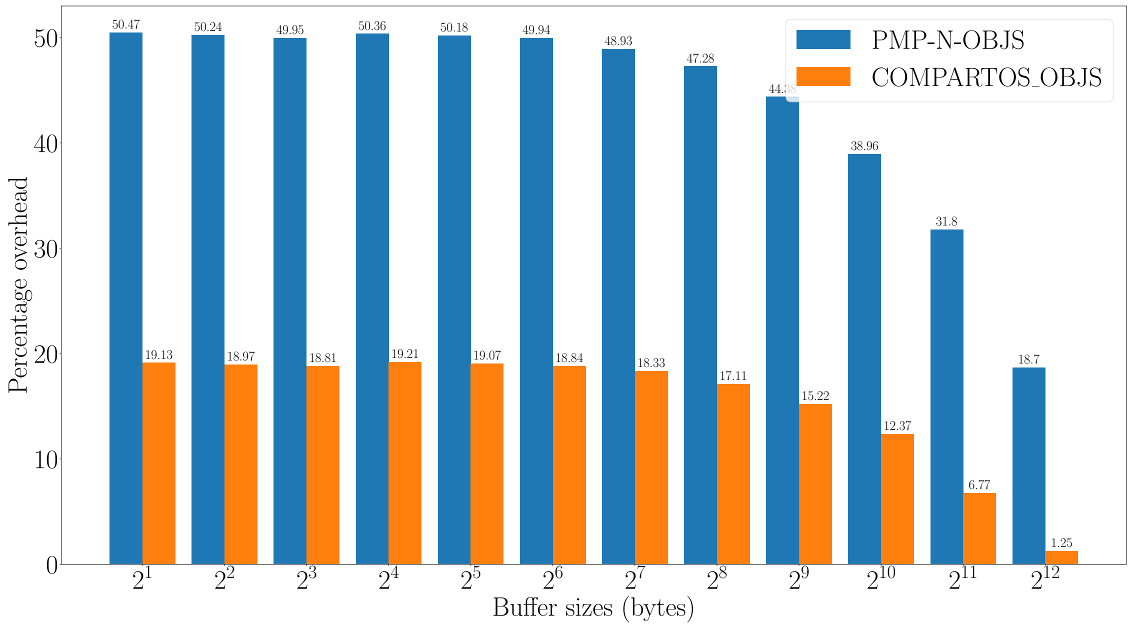


Figure 5.14: Overhead percentage of instructions count compared to INSECURE-STATIC baseline when varying buffer sizes to send a fixed-size data of 4 KiB. A non-CHERI-aware memcpy is used, doing byte-by-byte copies across all software variants.

5.11 CoreMark: CPU Performance Evaluation

CoreMark is an industry-standard benchmark that measures CPU performance [123]. It contains a suite of common algorithms such as sorting linked lists, matrix multiplications, and state machine operations. Such operations are frequently used in embedded applications. We use CoreMark as a performance macrobenchmark, representing a real-world, compute-intensive workload. Unlike microbenchmarking, evaluating performance using CoreMark amortises the performance overheads across different software variants, thus, helping analyse the performance impact of different protection models in a more realistic workload. That said, there is no security benefit compartmentalising CoreMark by itself as it is a single application without security boundaries between its benchmark suites and files. Still, it helps us analyse where performance is consumed in similar applications that perform similar operations to CoreMark while actually having some security boundaries. For example, the below results show how PURECAP's doubled register size affects caches and how CompartOS' GPREL addressing affects instruction count, if at all.

We ported CoreMark to run on FreeRTOS but also configured the benchmark to support CHERI pointers. We set the number of benchmark iterations to 8000 to evaluate the performance effects of CompartOS on similar intense workloads. CoreMark, by design, is required to run for at least 10 seconds to ensure consistent results with relatively low standard deviations. The 8000 iterations number was chosen accordingly. We found out that the standard deviation between different runs is very close to zero with that number, and increasing it does not change the CoreMark score. CoreMark reports a single output number as a standard score to be compared against. This is the number of performed iterations per second, divided by the processor frequency and is often referred to as CoreMark/MHz.

In the **PMP-N-OBJS** and **COMPARTOS-OBJS** variants of CoreMark, there are 6 object compartments, and the number of resources attached to each ranges between 8 to 72. Frequent domain switches happen over the benchmark period. Also, the benchmark is single-threaded, so **PMP-4-TASKS** is the same as the unprotected baseline variant (INSECURE-STATIC).

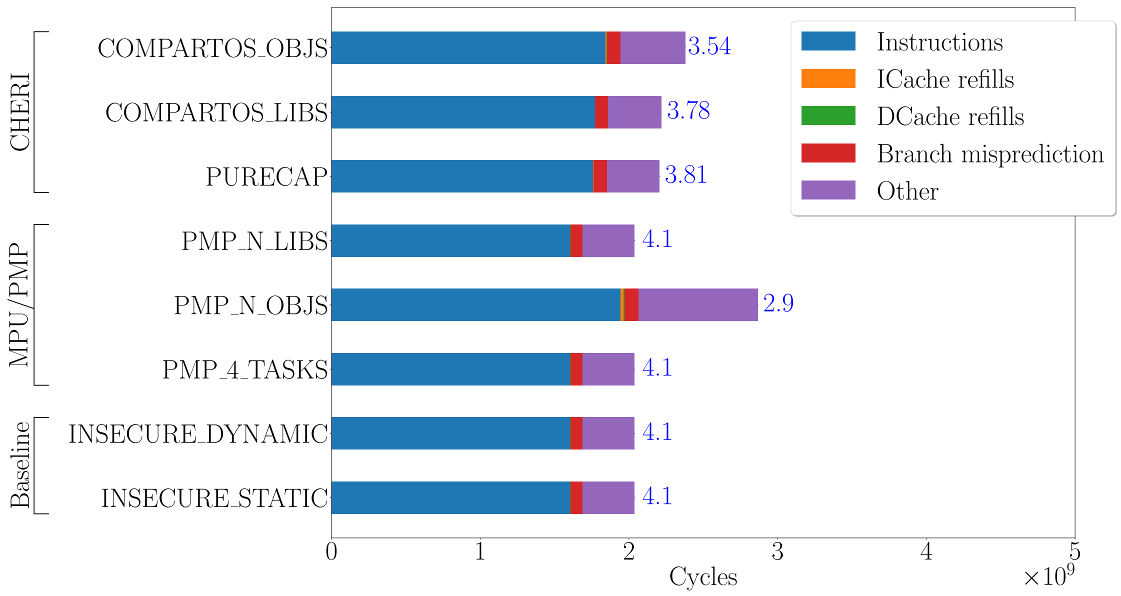


Figure 5.15: Performance of CoreMark in cycles. The labelled numbers beside each bar are the standard CoreMark scores which is the number of performed iterations per second, divided by the processor frequency. The bigger that number is, the better. As mentioned, we run with 8000 iterations and 100 MHz processor frequency.

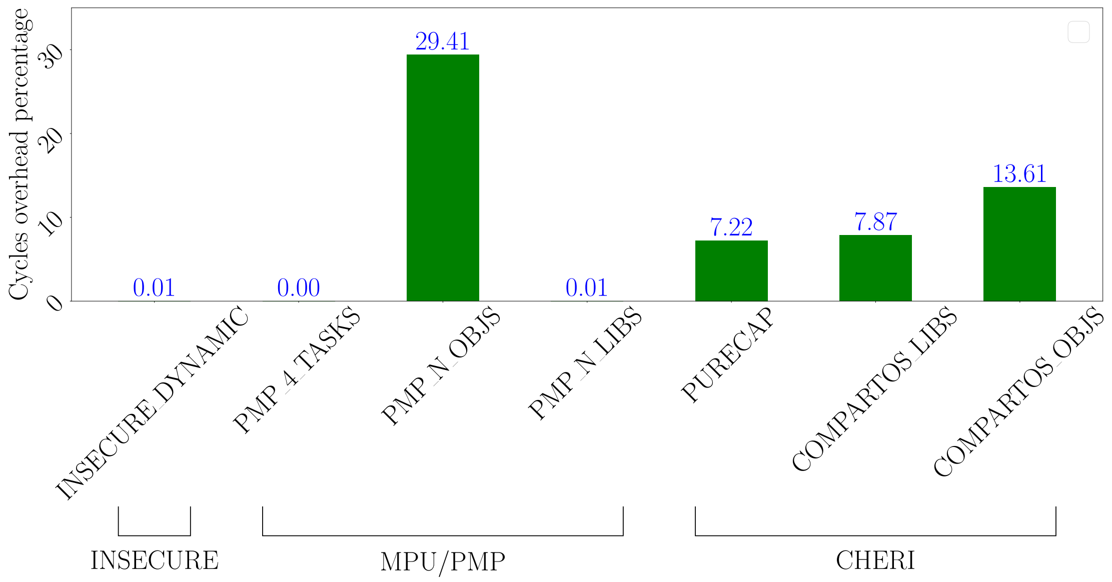


Figure 5.16: Cycles overheads of CoreMark compared to INSECURE-STATIC baseline.

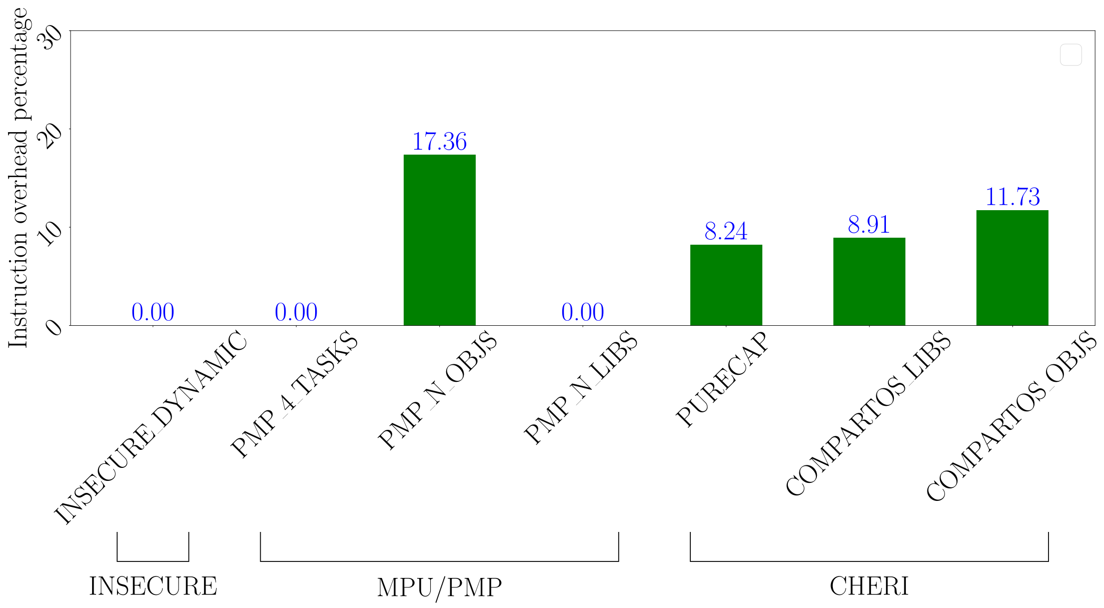


Figure 5.17: Instructions count overheads of CoreMark compared to INSECURE-STATIC baseline.

Figure 5.15 illustrates the absolute number of cycles taken for each variant, while Figures 5.17 and 5.16 show the instructions and cycles overheads, respectively, compared to **INSECURE-STATIC** baseline.

As expected, dynamic linking and loading offered by `libdl` do not incur any performance overhead. **PMP-N-OBJS** adds 29.41% overhead compared to **INSECURE-STATIC** while **COMPARTOS-OBJS** only adds 13.61% overhead.

Conclusion: In our FreeRTOS/CoreMark benchmark prototype, CompartOS is 44.7% faster than MPU-based object compartmentalisation (e.g., in ACES).

Compared to baseline, **PURECAP** has an 8% overhead in performance. As shown in Figure 5.15, this is due to extra generated instructions to manipulate capabilities which also affect instruction cache misses.

COMPARTOS-LIBS does not have any domain switches as the entire application is built as a single library compartment. Thus, the difference between **COMPARTOS-LIBS** and **PURECAP** is basically the GPREL addressing cost. Based on the CoreMark scores, there is not a significant overhead for CompartOS' GPREL addressing compared to non-compartmentalised PURECAP.

Conclusion: In our CheriFreeRTOS prototype, CompartOS' new GPREL addressing and partitioned captables incur negligible overhead compared to non-compartmentalised PURECAP.

5.12 TCP/IP Benchmarks: Performance, Security, and Compatibility Evaluation

We perform TCP/IP evaluation to understand how a real-world mainstream system would perform across the different protection models implemented by our FreeRTOS software variants. The evaluation includes response time, throughput, and security. As most feature-rich IoT systems do have a TCP/IP stack for communication and connectivity, this TCP/IP use case is the most critical, complex, and realistic application we are discussing so far. It is important to note that we evaluate mainstream TCP/IP centric applications that are deployed by FreeRTOS and are published publicly [124], including TCP/IP stack, a FAT filesystem, various servers and network protocols such as HTTP, FTP, TFTP, and CLI, as shown in Figure 5.18. That is, we have not developed it ourselves, and thus also we evaluate the compatibility and applicability advantage of CompartOS.

This use case has multiple software subsystems and libraries (represented as small boxes in Figure 5.18) that are mutually separate, often distrusting, and sometimes come from different vendors with different criticalities and real-world published vulnerabilities. Each library has a clear, documented, and well-defined logical API; communication between them is either via direct function calls or events and IPC. Thus, the security boundaries between such compartments are clear and realistic compared to the previous benchmarks we discussed. That makes this demo an ideal candidate for evaluating compartmentalisation. Figure 5.18 shows the automatically generated and compartmentalised graph for this benchmark that we use across the section. Each box in this figure is a separate logical component or library, and we enforced compartmentalisation between each by deploying linkage-based, library-based compartmentalisation.

First, we evaluate the response time of sending a single packet from a host PC to the FreeRTOS demo, which is a frequent operation in networking. For example, single packets are used in operations such as ping (to check the network speed or the liveliness of host devices), TCP acknowledgement packets, and one-packet embedded systems commands (e.g., to stop, run, or control a motor). Then we perform a throughput evaluation to amortise the overheads in single-packet pings by performing FTP upload operations. This involves more packets and

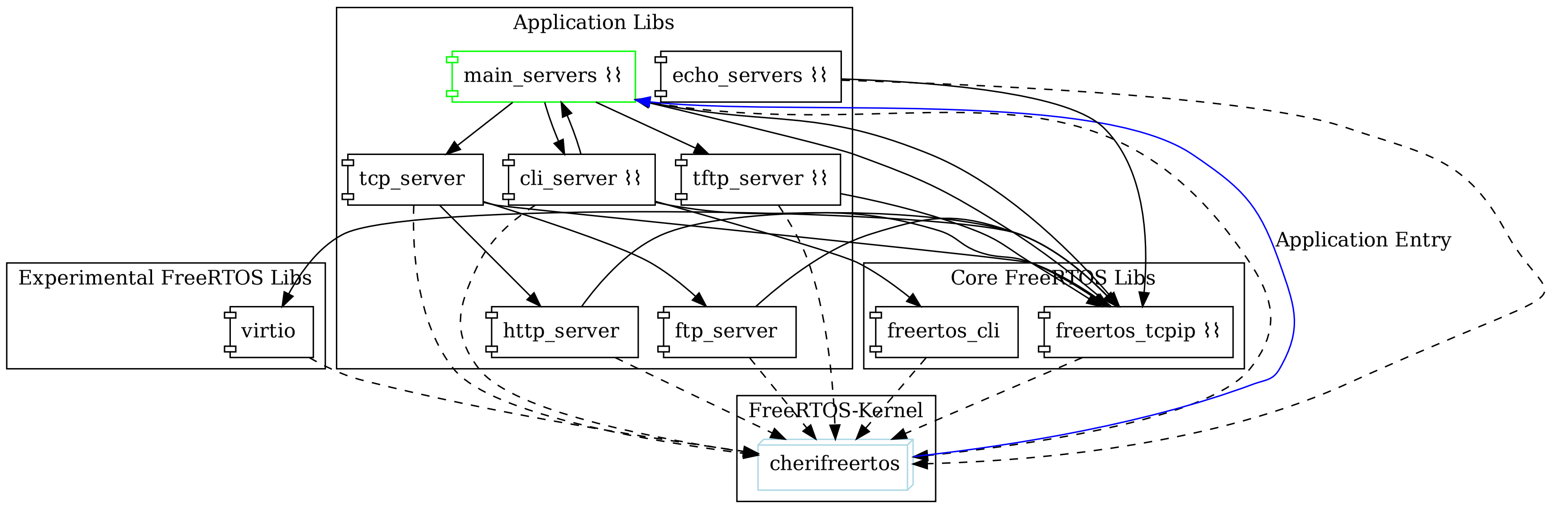


Figure 5.18: Compartmentalised TCP/IP demo graph that is statically and automatically generated by CheriFreeRTOS during the build stage. The figure is mapped to the COMPARTOS-LIBS variant that is linkage/library based compartmentalisation. Small boxes represent linkage-based library compartments, and the edges are the API references a compartment makes to another. Dotted edges indicate a trusted call to FreeRTOS without trampolines or domain switches. Compartments are placed into larger boxes to categorise them into core, third-party, and application libraries.

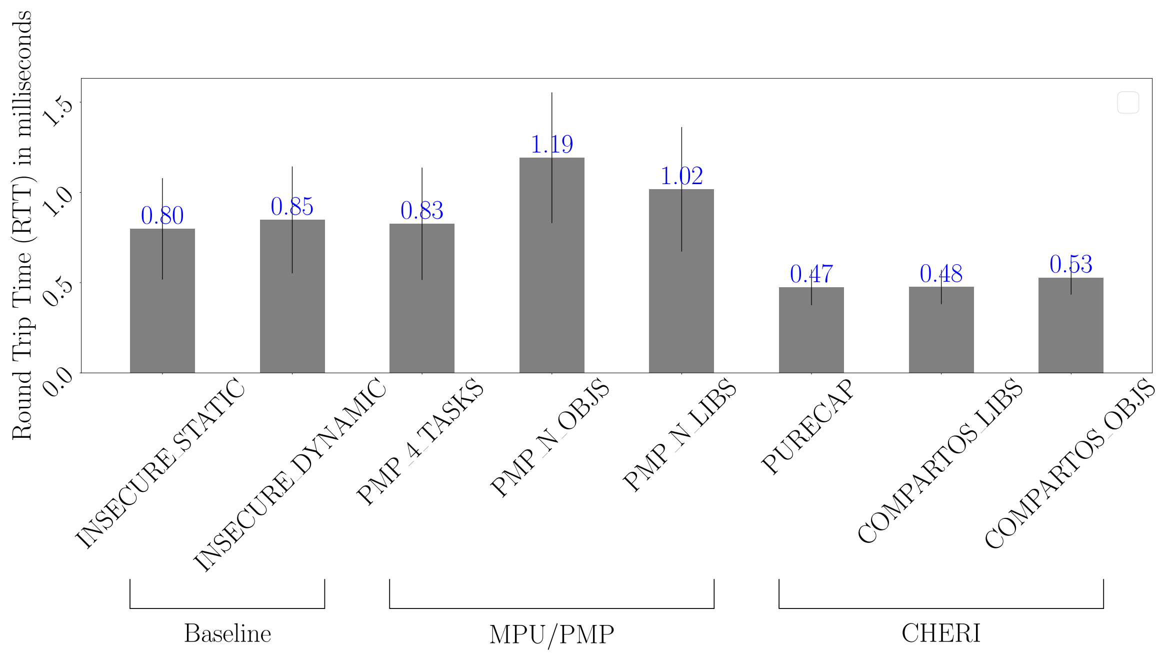


Figure 5.19: Small packet round-trip latency with ping (lower is better).

protection domain switches compared to sending a single packet. Finally, we perform a security evaluation by analysing the recent publicly published FreeRTOS TCP/IP stack vulnerabilities and how CHERI and CompartOS could protect a system with those exploitable vulnerabilities.

Our networking setup is a computer that is attached directly over a 1 Gigabit Ethernet (cross-over) to the VCU-118 board running FreeRTOS on our processor variants running at 100 MHz.

5.12.1 Response Time

We perform response time evaluation by sending a total number of 100 ping packets and getting the average and standard deviation numbers reported by the ping command on a Linux PC. The command calculates the standard deviation of the Round Trip Time (RTT), which is shown in Figure 5.19. Ping sends a single ICMP packet from the client to the server and receives back an acknowledgement packet. This makes the code path short and fast and thus maximises the overheads. Further, the ICMP packet processing only incurs one domain switch in the case of library compartmentalisation from the FreeRTOS ISR to the TCP/IP stack, as shown in Figure 5.18. The **PMP-4-TASKS** is also effectively doing nothing when it comes to protection; FreeRTOS-MPU does not try to rework the TCP/IP stack (being relatively complex and large) to use task-based MPU compartmentalisation as that is likely inadequate due to the MPU limitations mentioned so far.

By comparing **PURECAP** against the **INSECURE-STATIC** baseline, we find that the RTT is 62.66% faster, as shown in Figure 5.20. This is because CHERI doubles the register sizes, which halves the number of instructions spent in *memcpy* that is frequently used in the TCP/IP code paths. For example, *memcpy* is frequently used to send network events over FreeRTOS' queues (IPC) from the device driver to the TCP/IP stack and from a timer server to the TCP/IP stack. Those IPC events that use *memcpy* dominate the ICMP code path, and thus the CHERI memcpy advantage is maximised. We cannot use a non-CHERI-aware *memcpy* in this case,

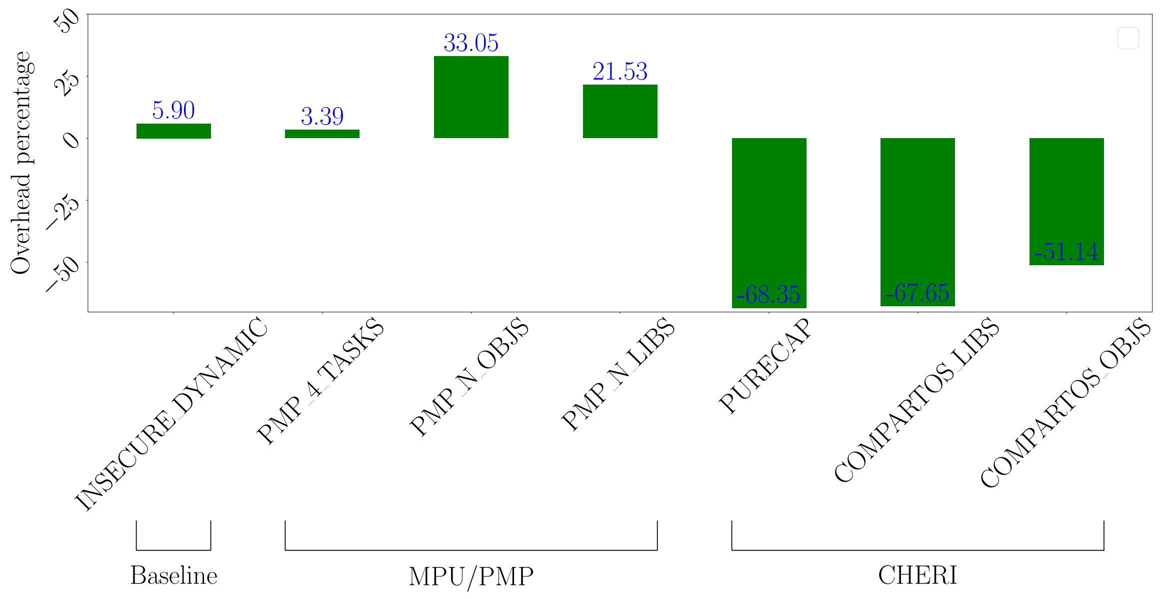


Figure 5.20: Small packet round-trip latency overheads compared to Insecure-baseline

as events and the IPC buffers include pointers. However, in Section 5.12.2, we amortise this advantage by using a non-CHERI aware *memcpy* used when copying many data packets that do not include pointers and dominate the throughput. Therefore, comparing CompartOS variants (that are inherently PURECAP) here against the PMP ones might be dominated by the *memcpy* advantage, so we do not try to directly compare them. That said, comparing the **COMPARTOS-LIBS** against **PURECAP** gets us the compartmentalisation and protection domain switches overhead. This is a negligible 3.6% for such a small operation. Similarly, if we compare **PMP-N-LIBS** against **INSECURE-STATIC**, we find that PMP-based compartmentalisation adds 25% overhead. A large part of this high overhead is because of the domain switch between FreeRTOS ISR and the TCP/IP stack, which is dominated by costly PMP reconfigurations. Finally, looking at the standard deviations, the non CompartOS variants all have over .3 milliseconds while CompartOS variants have .1 milliseconds or less. The reason is because of the reduced cache miss rate in CompartOS when copying buffers and sending events over queues, compared to the PMP variants. This suggests that CompartOS could be more deterministic for real-time systems.

In the next section, we measure the throughput rather than response time, which makes the protection and compartmentalisation overhead quite visible and dominant.

5.12.2 Throughput

In this section, we measure the throughput of different implemented models by uploading a file from the host computer to an FTP server running on FreeRTOS, as shown in Figure 5.18. This is the most realistic workload and resembles a real-world use case where there are different protocols, compartments, and subsystems triggering multiple protection domain switches. The aim is to investigate how different implementations of protection models affect the overall performance by amortising the cost of timer interrupts, memcpy advantage in CHERI, and single-packet processing time, discussed in the previous section. We replaced all of the *memcpy* calls that do not contain pointers to be all 4-bytes (the size of integer registers in our baseline 32-bit

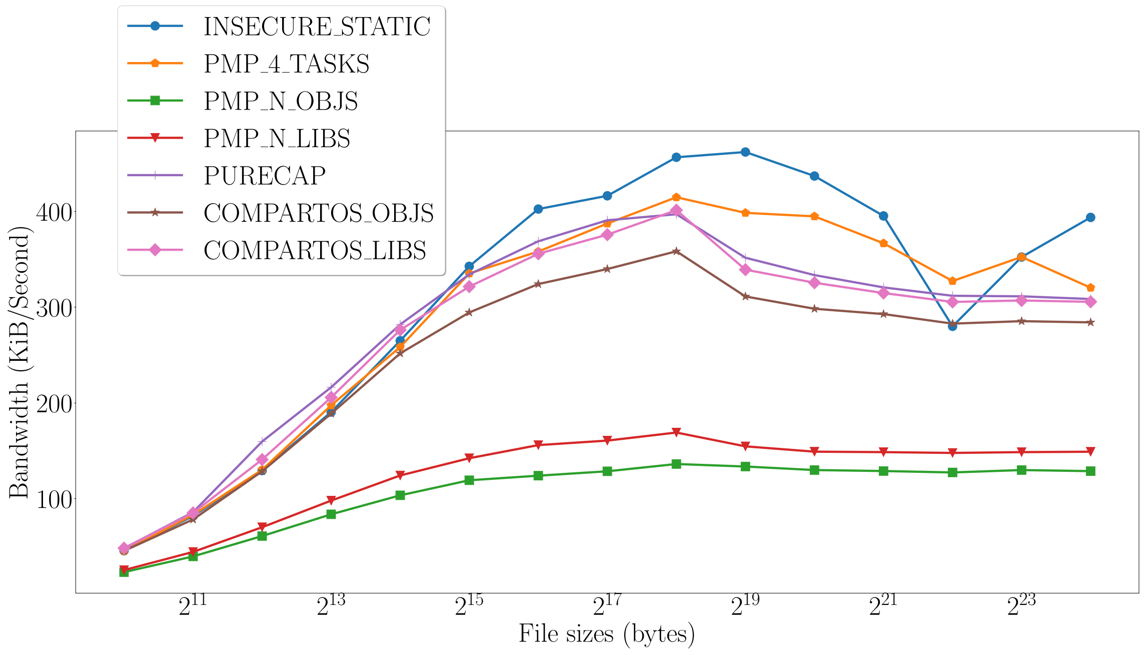


Figure 5.21: FTP upload bandwidth of varying file sizes.

Flute) across all variants, including PURECAP and COMPARTOS. For example, the payload of each packet that contains the file data are copied from the device driver to the network stack using 32-bit words instead of 64-bit CHERI capability registers. Similarly, filesystem-related reads and writes that are called while uploading the file are using non-CHERI-aware memcpy. For each received packet, there are multiple protection domain switches between task-based compartments, multi-threaded compartments, and library-based compartments. For instance, a packet is received by the FreeRTOS ISR, which jumps to the VCU-118 network device driver, which is part of a multi-task compartment (*freertos_tcpip*) as it contains a thread for the ISR and a thread for the TCP/IP stack processing. Communication between them is IPC-based over queues, thus triggering a context switch that implicitly performs a protection domain switch. Later on, other compartments like *tcp_servers* and *ftp_server* and the filesystem are called to process the FTP commands and write the file before sending back an acknowledgement and response packet to the host. Overall, the processing includes multiple domain switches, IO handling, memory-intensive operations, and per-packet and scheduling handling, thus demonstrating a rich and real diverse workload that is commonly used in deployed embedded systems.

Figure 5.21 shows the absolute bandwidth of uploading different file sizes from the host to FreeRTOS over FTP. The most relevant and important variants there are **COMPARTOS-LIBS** and **PMP-N-LIBS**, with library compartments represented in Figure 5.18. The bandwidth stabilises after a certain file size as the host PC cannot send packets any faster due to its hardware and Linux networking subsystems. This is shown by the fact that the FreeRTOS' idle task utilisation increases after the same file size, as shown in Figure 5.22. Figure 5.21 shows that the linkage-based PMP variants are quite slow compared to all other variants, including CompartOS and PURECAP, regardless of the file size. The bandwidth overheads further increase across the PMP variants when the file size increases. This suggests that PMP-based compartmentalisation is not as scalable as CompartOS.

In Figure 5.23, we pick a file size of 8 MiB when the bandwidth stabilises and compare the overheads against **INSECURE-STATIC** in Figure 5.24. As shown, the **PMP-N-LIBS**

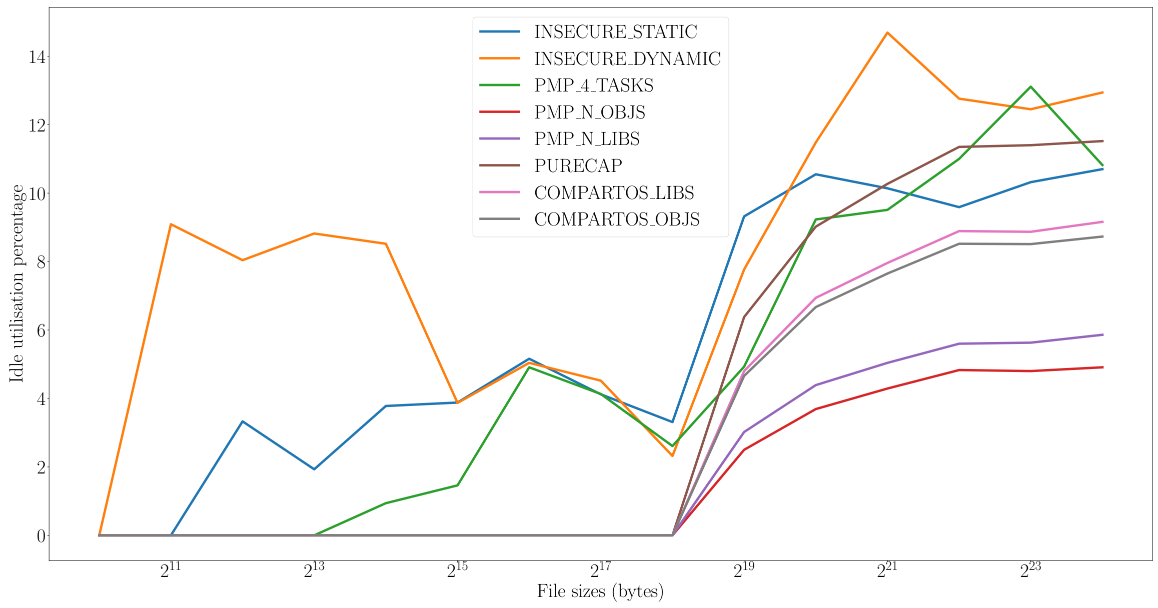


Figure 5.22: Idle task utilisation during FTP uploads of various file sizes.

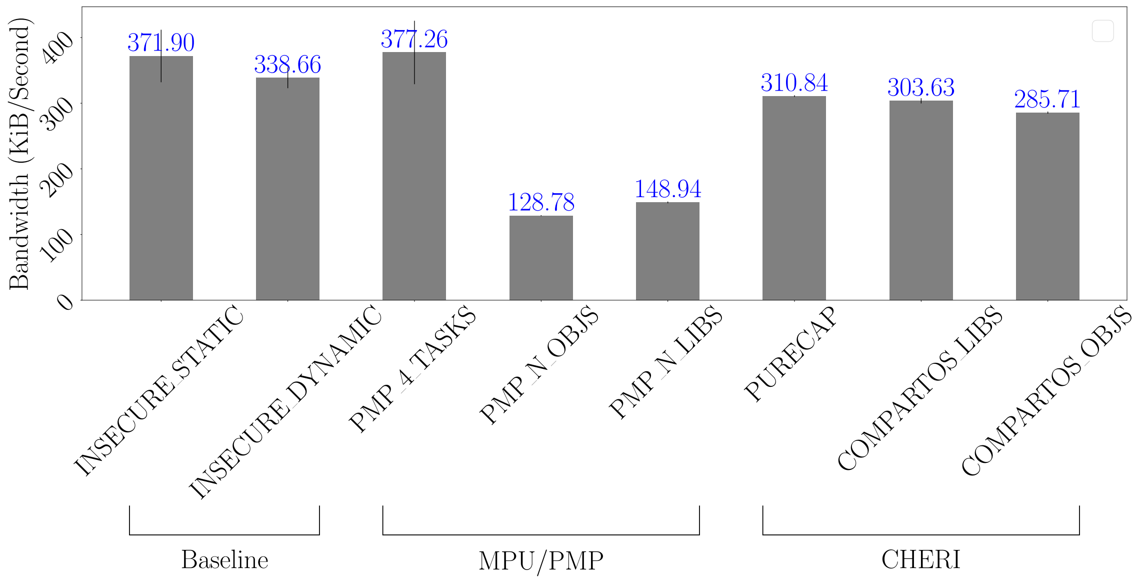


Figure 5.23: FTP upload bandwidth for 8 MiB file size.

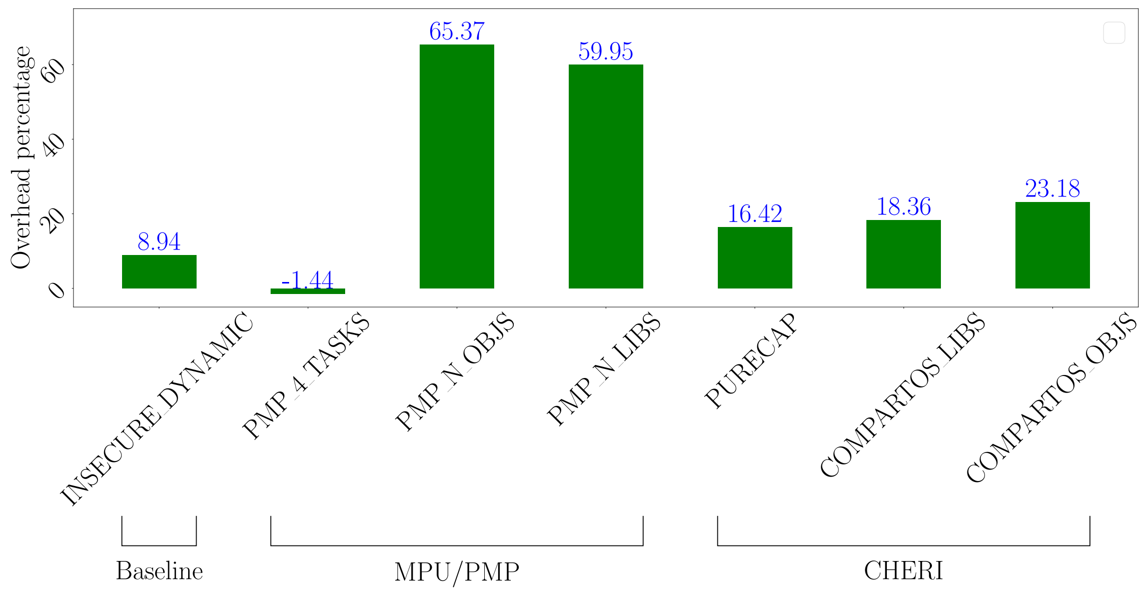


Figure 5.24: Overheads of FTP upload bandwidths for 8 MiB file size against INSECURE-STATIC baseline.

has a significant 60% overhead while **COMPATOS-LIBS** only incurs 18% overhead. Further, **COMPATOS-LIBS**'s overhead is less than 2% higher compared to **PURECAP**'s, which suggests that the compartmentalisation overhead (GPREL addressing, trampolines, and per-compartment captables) is negligible.

Conclusion: In our FreeRTOS prototype, CompartOS' TCP/IP bandwidth is 52% faster than linkage-based, MPU-based compartmentalisation.

5.12.3 Security

We evaluate the security aspects of the new compartmentalisation model by analysing the integrity, confidentiality, and availability of a compartmentalised FreeRTOS vulnerable system. This yields a good speculative indication about the nature and impact of future unknown vulnerabilities and how they can be caught by CHERI and compartmentalised by CompartOS. We use the National Vulnerability Database (NVD) [125] from the US government's National Institute of Standards and Technology (NIST) as a security metric system for assessing the severity and security implications of some public FreeRTOS Common Vulnerabilities and Exposures (CVEs). Table 5.8 lists thirteen public FreeRTOS network stack CVEs [126]. The severity column is the reported overall base score by NVD, which includes sub-scores of impact, exploitability and other attack vector scores. It also reports scores for confidentiality, integrity, and availability of each CVE, which are also included in the table.

Based on our analysis, we found out that at least 10 out of 13 CVEs would have been caught directly by CHERI, protecting against integrity and confidentiality vulnerabilities. In a high-confidentiality or -integrity system (e.g., past CHERI-based work that focuses on UNIX), that may be enough; however, in a high-availability, safety-critical system, it is not. For example, even in a non-compartmentalised CHERI system, catching integrity and confidently violations will

CWE Number	Description	Severity	Confidentiality	Integrity	Availability
CVE-2018-16528	AWS secure connectivity modules – mbedTLS context corruption	8.1 (High)	Yes	Yes	Yes
CVE-2018-16522	AWS secure connectivity modules – uninitialized pointer free	8.1 (High)	Yes	Yes	Yes
CVE-2018-16526	usGenerateProtocolChecksum memory corruption	8.1 (High)	Yes	Yes	Yes
CVE-2018-16525	DNS/LLMNR memory corruption/information leak	8.1 (High)	Yes	Yes	Yes
CVE-2018-16599	NBNS memory corruption/information leak	5.9 (Medium)	Yes	No	No
CVE-2018-16601	IP DoS/Memory corruption	8.1 (High)	Yes	Yes	Yes
CVE-2018-16523	TCP Options information leak/DoS	7.4 (High)	Yes	No	Yes
CVE-2018-16524	TCP Options information leak/DoS	5.9 (Medium)	Yes	No	No
CVE-2018-16603	TCP information leak	5.9 (Medium)	Yes	No	No
CVE-2018-16602	DHCP information leak	5.9 (Medium)	Yes	No	No
CVE-2018-16600	ARP information leak	5.9 (Medium)	Yes	No	No
CVE-2018-16527	ICMP information leak	5.9 (Medium)	Yes	No	No
CVE-2018-16598	DNS Poisoning	5.9 (Medium)	No	Yes	No

Table 5.8: FreeRTOS 2018 published vulnerabilities

trap or otherwise may crash the entire system. Therefore, the CompartOS compartmentalisation model takes security further to allow a compartment-specific handler to catch and handle the violation signal, ensuring the system can continue to operate—thus, meeting the improved availability design goal.

Conclusion: CHERI could detect 10 out of 13 FreeRTOS' TCP/IP stack CVEs (that CompartOS can recover from) that MPU-based, task-based compartmentalisation could not.

We have reproduced CVE-2018-16526 and developed an exploit for it as a proof-of-concept example for vulnerability mitigation and improved availability evaluation. This CVE is very severe and similar to other CVEs in the table (as far as reproduction and exploitation); it is also a base on which others can build. This vulnerability may affect both the confidentiality and integrity of the system by leaking information and, in some FreeRTOS configurations, allowing remote code execution. It exploits the fact that the TCP/IP stack did not check for the IP options field in packets while other code fragments still used the IP header length field. A carefully crafted packet triggers out-of-bounds buffer accesses (in the same TCP/IP compartment) which CHERI immediately catches. Thus, the confidentiality and integrity of the system are maintained. MPU-based compartmentalisation techniques cannot detect or recover from any of such vulnerabilities as they are all in the same single TCP/IP compartment.

Once caught, a fault-handling mechanism needs to be applied in order to maintain the availability of the TCP/IP stack in cases that may affect its integrity. We have deployed a few fault handling and recovery mechanisms that we discussed in Sections 3.3 and 4.4.3, and we evaluate them later in Section 5.14 within a safety-critical automotive demo. The TCP/IP stack keeps functioning and will not provide a DoS attack surface. Furthermore, CVE-2018-16525 and CVE-2018-16599 are similar to CVE-2018-16526—a crafted packet with a malicious protocol length field that mismatches the actual protocol payload length could cause out-of-bounds accesses that can be caught by CHERI. CVE-2018-16601 builds on CVE-2018-16526 (that we reproduced) and could cause DoS. It causes an integer overflow that may lead to accessing the entire address space in a subsequent *memmove* call. Since memmove arguments are capabilities rather than just integer addresses, if the passed buffer length is larger than either the source or destination buffers (in that case, those are the IP packets), a CHERI length violation will be immediately triggered. Similarly, CVE-2018-16523 and CVE-2018-16524 are vulnerable to malicious TCP options field in a packet that causes out-of-bounds accesses. There is also a potential divide-by-zero exception if the supplied MSS field is zero. While RISC-V does not trap in that case, compartmentalisation would help in other architectures that do to further handle such faults; simply dropping the packet would be sufficient. However, even non-CHERI exceptions (such as divide-by-zero) can be caught and handled in isolation without affecting the rest of the system by virtue of generic software compartmentalisation offered by CompartOS. This improves availability, especially with DoS-style attacks. CVE-2018-16602, CVE-2018-16600, and CVE-2018-16527 exploit the fact that some specific protocols (e.g., DHCP, ARP and ICMP) modify and truncate the received packets then send them back. If the received packet is not large enough to hold the to-be-sent packet, this can cause out-of-bounds reads that will be caught by CHERI. Except for CVE-2018-16522, CVE-2018-16522, and CVE-2018-16598, all of the CVEs build on CVE-2018-16526 (that we reproduced) or can be directly caught by CHERI; therefore, our recovery mechanisms apply.

5.13 Case Study: Amazon's AWS FreeRTOS OTA

Amazon's FreeRTOS OTA is a complex, industrial, IoT end-to-end application that is an obvious candidate for applying CompartOS. It performs advanced operations for software updates using encryption, code signing, and secure communications with the cloud. The demo integrates a handful of (third-party) IoT libraries such as MbedTLS (SSL/TLS) [127], MQTT (IoT messaging protocol), TCP/IP, filesystems, OTA, and encryption libraries. This increases the attack surface of the system due to the potential bugs, vulnerabilities, and exploits in each library.

We compartmentalised OTA to evaluate CompartOS' design goals, including improved availability and a high degree of compatibility with existing software. The latter was demonstrated by the fact that the OTA source that is over 100k LoC did not require modification for use with CompartOS.

Conclusion: CompartOS secured a mainstream and complex OTA application that is over 100 KLoC without any source-code changes at all, asserting the compatibility and scalability goals.

The former is discussed further below. OTA also provided an opportunity to demonstrate CompartOS' support for partial software updates, a novel feature for embedded systems.

Figure 5.25 shows the generated compartmentalised graph. CompartOS primarily helps isolate exploitable vulnerabilities in FreeRTOS libraries, such as the TCP/IP stack (as discussed in Section 5.12.3), and third-party libraries, such as MbedTLS (Arm), tinycbor (Intel [128]), and OTA (AWS [129]). In a non-compartmentalised system, such vulnerabilities could be used to crash the entire system or steal sensitive information, such as private keys or certificates. CompartOS reduces that possibility and improves the availability of the system. Partial software updates and restarts could further be used as fault handling techniques to recover a crashed compartment.

Since OTA requires mutual authentication and sensitive keys and certificates, we also improved confidentiality by placing the keys into their own C file called *aws_secrets* and building it as a separate compartment, as shown in Figure 5.25.

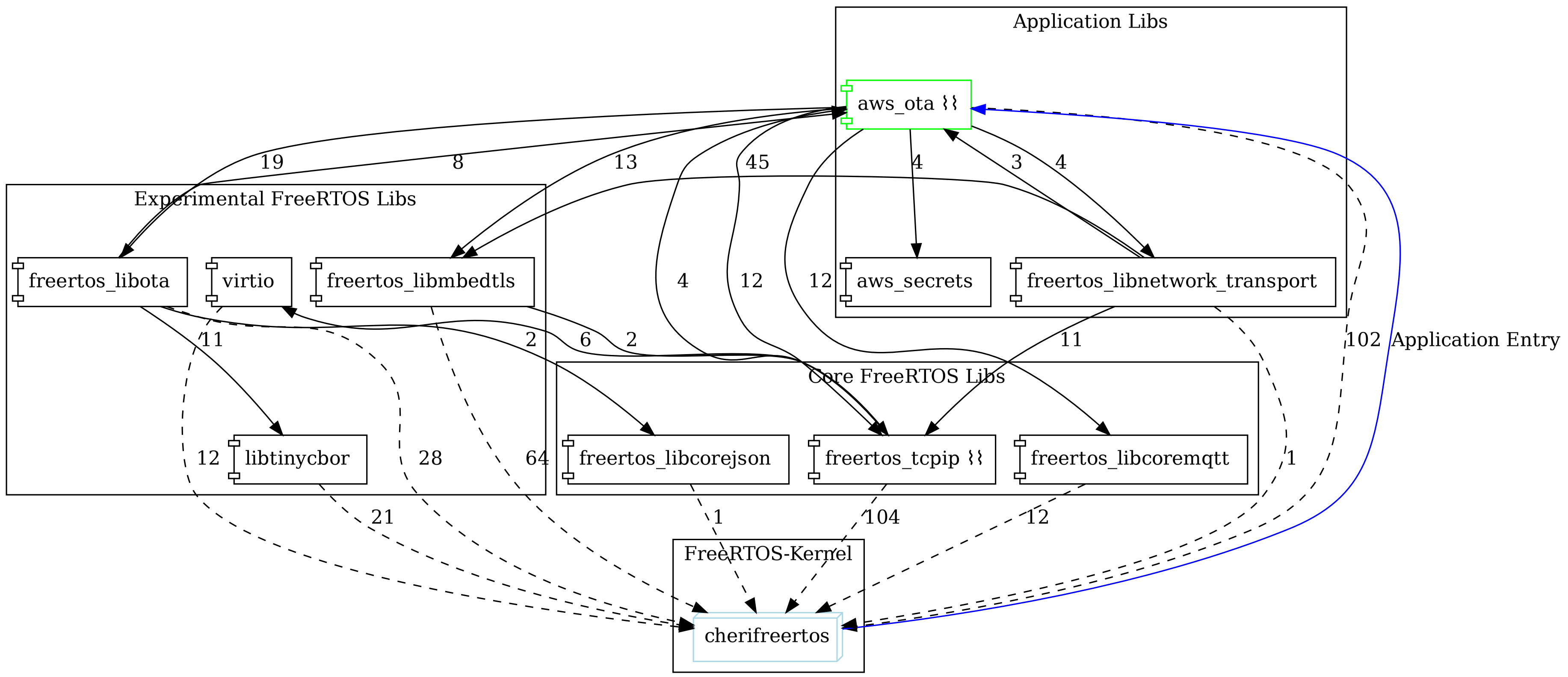


Figure 5.25: Amazon's Over-The-Air generated compartmentalised graph that is statically and automatically generated by CheriFreeRTOS during the build stage. Small boxes represent linkage-based object compartments, and the edges are the API references a compartment makes to another. Dotted edges indicate a trusted call to FreeRTOS without trampolines or domain switches. Compartments are placed into larger boxes to categorise them into core, third-party, and application libraries.

5.14 Case Study: Evaluating Security and Availability in a Safety-Critical Real-time, Automotive ECU

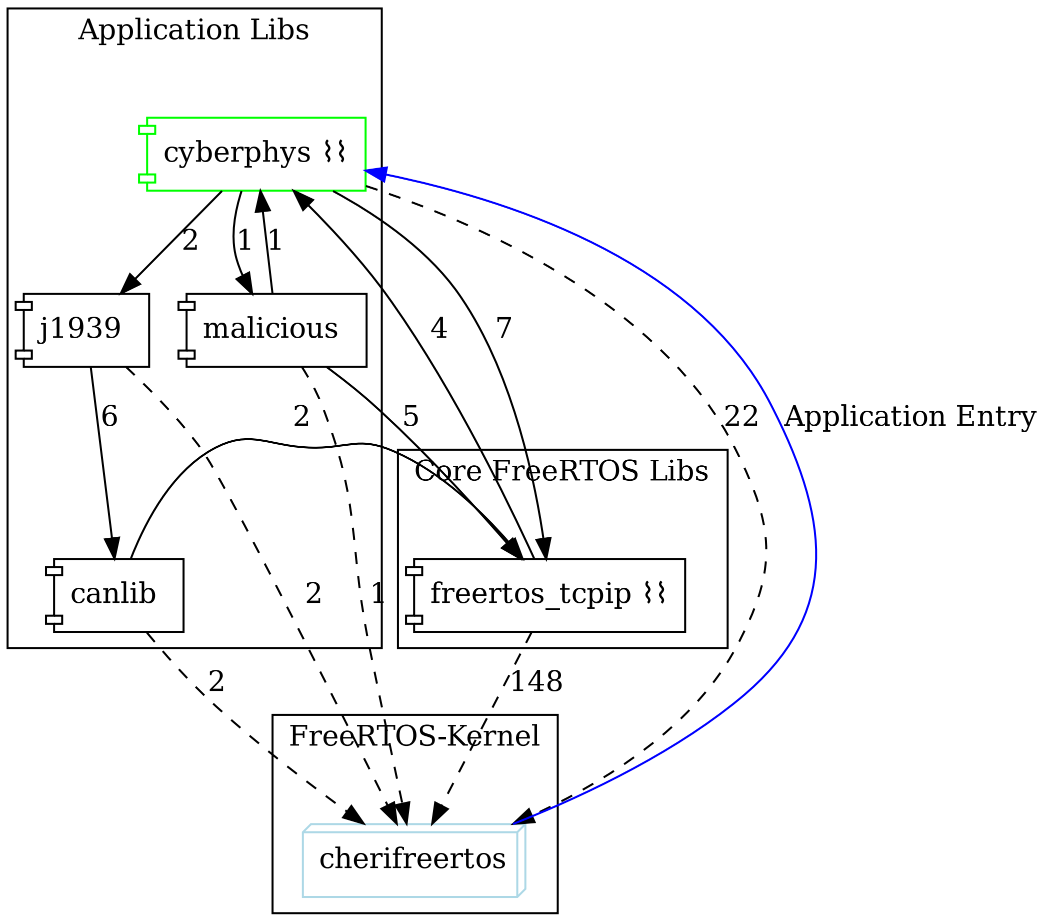


Figure 5.26: Automotive ECU compartmentalised graph that is statically and automatically generated by CheriFreeRTOS during the build stage. Small boxes represent linkage-based object compartments, and the edges are the API references a compartment makes to another. Dotted edges indicate a trusted call to FreeRTOS without trampolines or domain switches. Compartments are placed into larger boxes to categorise them into core, third-party, and application libraries.

The Automotive ECU demonstrator is an existing FreeRTOS-based system (not created by us) on which we apply the CompartOS model to evaluate the following goals of a real-time safety-critical system:

- Integrity: By catching buffer-overflow attack attempts due to CHERI spatial memory safety guarantees.
- Availability: By handling the buffer-overflow violations in a few microseconds and isolating the effects to only the vulnerable compartment by virtue of CompartOS' compartmen-

talisation.

- Compatibility: By not requiring any source-code changes.

The ECU demonstrator imitates a modern car, where multiple microcontrollers are responsible for various tasks and communicate with each other over a CAN network [130]. This ECU runs FreeRTOS and contains a buffer-overflow vulnerability in the SAE J1939 [131] protocol stack. An attacker could make use of that and carefully craft a J1939 packet that leads to a buffer overflow and remote code execution (e.g., disabling reading the brake pedal position).

Figure 5.26 illustrates the generated compartmentalised system architecture of the CompartOS Automotive ECU. Weighted edges are inter-compartment API references; each box is built into its separate library compartment. The green "cyberphys" compartment is the main application. As shown, there are direct dependencies between all of these components. If, for example, the j1939 compartment stopped functioning for a few seconds, the application would not be able to send brake commands to the brakes' ECU, a life-threatening scenario.

5.14.1 Evaluating Fault Handling

The ECU demo is a good use case for evaluating fault handling techniques discussed in Section 3.3 as it is a safety-critical system that is also subject to real-time requirements. It does integrate both critical and less critical compartments with different attributes and requirements covering all scenarios discussed in Section 3.3. There are three main compartments subject to attacks and fault handling:

- **Malicious**: representing an actively malicious, non-critical compartment trying to compromise the system or cause DoS.
- **J1939**: representing a vulnerable critical application compartment, subject to exploits.
- **freertos_tcpip**: representing a complex and vulnerable core OS library that both critical and non-critical compartments depend on.

We evaluate fault handling in the three compartments and report the lesson and conclusions learned for each scenario.

5.14.1.1 J1939: Fault Handling of a Vulnerable Safety-Critical Compartment

The J1939 compartment is safety-critical and needs to be reliable and running all the time. It is used to send brakes and steering commands with real-time requirements; the main loop delay is ***50 milliseconds***; thus, commands should be detected and handled in less than that.

Return-Error: We experimented with a per-compartment fault handler that simply drops the malicious command and returns to the caller with an error. The entire fault handling and return take approximately ***30 microseconds***, which will not have any effect on the car's steering logic; therefore, the real-time guarantees are maintained. On the other hand, a non-compartmentalised system may be forced to perform a full restart as a brute-force recovery mechanism, which takes at least 2 seconds (the boot time of FreeRTOS and the Automotive ECU application)—an unacceptable latency for a safety-critical car brake system. However, Return-Error handling could cause memory exhaustion and DoS if the attack is recurrent as some buffers are allocated but not freed in that case.

Conclusion: In our CheriFreeRTOS/ECU demo, CompartOS' Return-Error fault handler meets real-time requirements of a safety-critical component, but might lead to DoS or memory exhaustion if the error is recurrent.

Custom-Handler: To address the issue of DoS incurred in Return-Error, we implemented a Custom-Handler, which requires knowledge of the design and implementation of the compartment itself, and assumes a faulting compartment would return an error code. It then takes further actions to free the buffer itself, thus not causing DoS or memory exhaustion. This is a more robust and secure strategy, but it requires a good knowledge of the underlying compartments, potential faults (to TRY-CATCH), and adds some implementation efforts. The custom handler for J1939 took 10 LoC changes that are mostly a new function to free the allocated canlib's buffer. The overall fault handing took ***60 microseconds***, which includes an addition of 30 microseconds to free the buffer, which is completely acceptable and meets the real-time requirements.

Conclusion: In our CheriFreeRTOS/ECU demo, CompartOS's Custom-Handler meets real-time requirements of a safety-critical component at a minimum development/knowledge effort.

Micro-reboot: The J1939 compartment has some attributes that made it possible to be micro-rebooted while meeting real-time requirements. The compartment is stateless; it only performs services, allocates buffers, processes and decodes the J1939 packets and finally returns. No other compartments rely on it having a specific state, and it does not pass buffers or contexts to dependent compartments (e.g., giving sockets in the TCP/IP stack to callers or returning memory allocated buffers to applications). Further, it is small and simple enough to perform a snapshot of its ELF sections and roll them back to a known start state on faults, effectively performing a micro-reboot. The section sizes of the compartment are as follows:

- captable: 896 bytes.
- text: 2609 bytes.
- data: 16 bytes.
- bss: 32 bytes.
- rodata: 32 bytes.

Since text and rodata can not be written (due to CHERI permissions), only bss and data need to be rolled back to a start state. The process of doing so takes ***83 microseconds***, which also meets the real-time requirements. No code changes were required at all. Further, we have also measured the time to roll back all the mentioned sections (though it might be unnecessary in most cases), and that takes 220 microseconds.

Conclusion: In our CheriFreeRTOS/ECU demo, CompartOS's micro-reboots could work fine for relatively small and simple compartments that might be stateless, while meeting the compatibility requirement.

5.14.1.2 Malicious: Fault Handling of a Non-Critical Actively Malicious Compartment

The *Malicious* compartment represents a non-critical compartment that might have been dynamically loaded or taken control of. This threat model is not addressed in most secure embedded systems like TockOS and ACES. This compartment could be something like a web browser, FTP server, or music player; all are non-critical to the car steering system but may have zero-day vulnerabilities that might be exploited. Malicious could try to attack confidentiality and integrity, and that will be prevented by virtue of capability-based security (i.e., not being able to forge a capability to access other resources) offered by CHERI/CompartOS. More importantly, Malicious could try to perform DoS attacks by faulting frequently or performing recursive calls, thus preventing critical compartments from executing or affecting their real-time requirements.

Most of the fault handling techniques could be applied here as Malicious is non-critical and does not have critical compartments relying on it. However, in such a scenario, the compartment could be simply killed or suspended for good until further appropriate action is performed (e.g., software updates to fix the bug or remove/replace the entire compartment).

Compartment-Kill We injected the Malicious compartment with a faulting instruction and evaluated Compartment-Kill fault handling. As discussed in Section 3.3, killing a compartment is as straightforward as invalidating its root capability table register (CGP). Malicious tries to create sockets and send data over a port. The first caught violation attempt takes 14.4 milliseconds which may affect the real-time requirements of the main loop as it is part of its path. Once caught by CHERI, the entire fault handling technique (by CompartOS) takes ***22 microseconds*** which meets real-time requirements. Further attempts to invoke the compartment immediately fault as CGP is invalid, and that takes 22 microseconds or less. This prevents the Malicious compartment from accessing any external or internal functions or data capabilities and protects against DoS attempts. Finally, this technique did not require any source-code changes at all on the application level, thus maintaining compatibility.

Conclusion: In our CheriFreeRTOS/ECU demo, CompartOS's Compartment-Kill fault handling can prevent integrity and confidentiality attacks in non-critical compartments while meeting real-time requirements, maintaining the availability of the remaining system.

5.14.1.3 TCP/IP: Fault Handling of a Complex Compartment With Dependencies

The TCP/IP stack is the most complex compartment in this demo and likely most other RTOSes as well. It does have many resources, dynamically allocates memory, integrates network device drivers, and has many dependencies on it. Thus, a single vulnerability in the TCP/IP stack can massively affect the availability and overall security of the system. We have reproduced CVE-2018-16526, discussed earlier, and demonstrated a real attack that exploits it to understand the implications associated with fault handling in such scenarios.

First, in a non-secure system, this vulnerability could cause integrity and confidentially violations (e.g., remote code execution), which might take control of steering the car or crash it. Second, in a PURECAP system that is non-compartmentalised, the buffer overflow will be detected by CHERI, but this will trigger an exception and put the entire system on halt, thus affecting its availability and causing a crash.

CompartOS is uniquely able to identify the faulting compartment, its boundaries, and dependencies which allows it to take further fault handling actions. Only a custom fault handler (User-level Custom-Handler) could work for the TCP/IP stack as opposed to other fault handling techniques discussed in Section 3.3:

- Compartment-Kill: cannot be applied as other critical compartments rely on it.
- Micro-reboot: cannot be used, as other compartments may still use its resources (e.g., Sockets).
- Return-Error: cannot simply return an error, as there is no caller compartment (the attack starts by an Ethernet interrupt that directly invokes the TCP/IP device driver), and some buffers have already been allocated.

Thus, we chose to experiment with a custom fault handler that restarts the TCP/IP stack. The following implementation actions were required:

1. Reset some of the TCP/IP stack state, like global variables, semaphores, network status.
2. Free some memory resources and kill threads created by the compartment.
3. Notify the dependent compartments that the TCP/IP stack is going to be restarted.
4. Dependent compartments need to stop using the sockets and the TCP/IP compartments.
5. Reinitialise the TCP/IP stack.
6. Dependent compartments need to recreate sockets and can resume using the TCP/IP compartment.

Each compartment thus has:

- (Re-)start: an entry point for restartable compartments.
- Kill: Free all resources it holds and deny access to itself.
- Stop: Temporarily deny access to itself by other compartments.
- Dependents-callbacks: Registered callbacks for each dependent compartment called when this compartment is being killed, stopped, or restarted.

This mimics some form of Object Oriented Programming (OOP) having constructors, destructors, garbage collection, and a chain of dependencies to handle.

Conclusion: CompartOS model allows further OOP fault handling techniques in complex compartments with rich resources and dependencies that are not possible in other task-based and MPU-based systems.

Such a process could be very time consuming and may violate some real-time requirements in some cases. In our evaluation, it took 1 second to perform a complete restart (in the background), while other critical components such as *cyberphys* kept functioning and meeting their real-time requirements. We compare that against non-compartmentalised **INSECURE-STATIC** and **PURECAP** systems that take at least 2 seconds to perform a complete restart of the whole system that stops all safety-critical compartments.

Conclusion: In our CheriFreeRTOS/ECU demo, CompartOS maintained availability of safety-critical real-time compartments while restarting a complex compartment with dependencies.

5.15 Summary

We have demonstrated how CheriFreeRTOS (a prototyped implementation of the CompartOS model) would perform compared to insecure baseline and other state-of-the-art security architectures by implementing their models in FreeRTOS. Besides complete pointer safety offered by CHERI, CompartOS could uniquely compartmentalise a wide range of embedded applications. Section 5.5 started by looking at the hardware cost of adding CHERI support to an embedded processor and compared that to other MPU/PMP, and MMU costs. We found that CHERI incurs an area overhead but is comparable to PMP and MMU, while it adds an unparalleled security benefit. We then performed a software evaluation to analyse performance, security, compatibility, and applicability in real-world embedded systems. Section 5.9 showed that it takes minimum effort in terms of LoC changes to support CHERI and CompartOS in an embedded OS such as FreeRTOS. In return, applications and libraries on top of OSes could entertain complete pointer safety and automatic software compartmentalisation using CompartOS, without source-code changes at all. This was demonstrated across large and complex use cases and benchmarks. We have evaluated the performance overheads across a range of micro and macro benchmarks and real-world safety-critical use cases. While CompartOS adds lightweight overheads compared to an insecure baseline, it has outperformed other MPU/PMP implemented models that provide a comparable level of security and compartmentalisation. Finally, as embedded systems are mostly application-oriented, we have demonstrated three main uses cases, including a TCP/IP demo, a $\sim$100 KLoC OTA demo, and a safety-critical automotive car demo. We performed security analysis using NVEs while experimenting with various fault handling techniques and how they affect the timing and other critical compartments. We found out that CompartOS largely simplifies and minimises the efforts needed to secure such mainstream and critical applications while being able to maintain real-time requirements performing fault recovery.

CHAPTER 6

RELATED WORK

This chapter discusses a few security architectures that have been proposed for embedded systems. In particular, we describe how past and related software compartmentalisation approaches are similar to CompartOS and, when applicable, how CompartOS might be an improvement over them. In Chapter 2, we have given a general background on MPU and CHERI software compartmentalisation. In Chapter 5, we evaluated a few MPU, CHERI, and CompartOS prototypes in FreeRTOS. In this chapter, we list specific research and industrial systems that are using the MPU and CHERI to enforce advanced software compartmentalisation approaches in embedded systems. We also trade-off their design, implementation, and evaluation decisions against our requirements in Chapter 3 and our CheriFreeRTOS implementation (see Chapter 4) and evaluation (see Chapter 5).

Table 6.1: Security properties of related embedded security architectures.

System	Hardware	Security Properties	Security Limitations	Protection Domain	Resources	Domain Switches
ACES	Arm MPUs.	Source-code files and IO-based coarse-grained automatic compartmentalisation.	MPU hardware regions counts. Hardware traps for domain switching. No OS support. Cannot compartmentalise libraries. No availability.	Source-code files. IO-based compartments.	Memory regions, IO.	System calls, reconfigure MPU.
MINION	Arm MPUs.	Task-based, coarse-grained automatic compartmentalisation.	MPU hardware regions counts. Hardware traps for domain switching. Cannot compartmentalise libraries. No availability.	Tasks.	Memory regions, IO, kernel, and application tasks.	System calls, reconfigure MPU.
uVisor	Arm MPUs.	Task and linkage-based manual compartmentalisation. Focus on heap memory security. Dynamic loading.	MPU hardware regions counts. Hardware traps for domain switching. No compatibility. Reliance on an OS. No availability.	Source/object files, tasks, interrupt handlers.	Memory regions, IO, kernel, and task applications.	System calls, reconfigure MPU.
TrustLite/TyTan	Intel's EA-MPU.	Task and linkage-based manual compartmentalisation. Dynamic loading.	MPU hardware regions counts. Hardware traps for domain switching. No compatibility. No availability.	Source/object files, libraries, tasks, interrupt handlers.	Memory regions, IO, kernel, and task applications.	System calls, reconfigure MPU.
CheriRTOS	CHERI-MIPS.	Task-based, manual compartmentalisation, protecting data and code segments of tasks, and spatial heap memory.	Manual compartmentalisation. Cannot represent libraries or OS subsystems not using tasks. No availability.	Tasks.	Memory regions, IO, kernel, and task applications.	Task switches (IPC). Reloading capability register file per task. Direct user-level IPC using CHERI's sealing mechanism and a custom domain switch instruction.
CheriFreeRTOS	CHERI-RISC-V.	Linkage-based automatic compartmentalisation, complete spatial memory safety and spatial compartment isolation. Focus on availability and recovery. Dynamic loading.	No temporal safety (but can easily be supported).	Source/object files, tasks, libraries, interrupt handlers.	Pointers, symbols, system registers, memory regions, IO, kernel, and applications.	Fast inter-compartment function calls, reloading a single root capability table register. Task switches (IPC).

6.1 ACES: Automatic Compartments for Embedded Systems

ACES [26] is an MPU-based software compartmentalisation approach that aims to automatically create compartments at build time by statically analysing the source code and an input security policy. It relies on off-the-shelf MPUs to enforce memory protection and instruments the final binary with MPU-based compartment switches.

A protection domain in ACES is deduced and created based on an input user policy, functionality, and data/code dependencies for each compartment. The resources ACES protect are merely static code and data sections and peripherals for each compartment. That is, there is no runtime resources (e.g., dynamic memory) protection or isolation.

As ACES compartments are file or IO based and not designed around tasks or processes (i.e., not task-based), existing baremetal software can be easily compartmentalised, thus achieving source-code compatibility. Moreover, there is no secure-monitor or hypervisor entity that intervenes in the inter-compartment operation, so it tends to be memory and performance efficient.

Due to the MPU hardware limitations, ACES sets a limit on the number of compartments which ranges from 13 to 34. Furthermore, to allow finer-grained protection beyond the MPU region granularity, ACES provides software emulation as a workaround, which could take a performance hit and make ACES impractical for systems in need of fine-grained isolation with a large number of compartments. For instance, ACES introduces a micro-emulator for stacks that cannot be allocated with the underlying MPU's alignment and size requirements, in which case unaligned memory access are trapped and emulated.

The MPU limitations further complicate the ACES compartmentalisation design and implementation. To also deal with MPU drawbacks on the number of compartments and performance overhead, ACES applies a complex merging strategy of regions that may expose data and peripherals to other compartments that may not use them.

CompartOS not only avoids all the MPU limitation workarounds that are thoroughly described by the ACES authors but also provides dynamic fine-grained capability-based security at a lower performance overhead, without compromises to performance or security. Further, unlike CompartOS, ACES does not try to integrate with, support, or discuss OS subsystems such as exceptions, threading, dynamic memory allocation, or fault handling.

However, ACES is similar to CompartOS in areas like specifying compartments as source-code files and feeding a security policy at build time. Even though ACES does not support dynamic loading and linking like in CompartOS, we believe that the CheriFreeRTOS' **PMP-N-OBJS** variant that we implemented and evaluated in Chapter 5 represents ACES at runtime (in performance, scalability, security, scalability) after the compartmentalisation process is completed, either at build-time in ACES, or runtime in CompartOS. Further, ACES cannot represent libraries like in our CheriFreeRTOS' **PMP-N-LIBS** variant. For example, ACES cannot compartmentalise large and mainstream embedded applications, such as the OTA, ECU, or TCP/IP demonstrators that we evaluated on top of CheriFreeRTOS, given that they are designed around tasks and (third-party) libraries, and not IO or source-files.

6.2 MINION: Securing Real-Time Microcontroller Systems through Customized Memory View Switching

MINION [36] is an MPU-based software security architecture for embedded systems that provides memory isolation between processes while optimising performance by avoiding performing frequent systems calls that are often associated with multi-privilege rings and MPUs. The paper argues that frequent system calls usually violate real-time system constraints and responsiveness.

Unlike ACES, MINION relies on a trusted high privileged software called a “view switcher” that restricts the memory view of each compartment, including the RTOS itself. Thus, RTOSes and user applications work side by side in the same unprivileged ring, while the “view switcher” acts like a monitor or a hypervisor. Like ACES, MINION performs static analysis of the embedded software/firmware to identify what memory regions each compartment needs. Compartment switches require a trap to the view switcher, which performs a protection domain switch using the MPU. Similarly, high privileged operations such as configuring the processor (e.g., to globally disable or enable interrupts) require system calls to MINION. MINION relies on off-the-shelf MPUs to enforce memory isolation using an ACL; the evaluation is done on a Cortex-M4 Arm processor with only 8 MPU regions. The authors recognise the scalability problem in using the MPU and try to cluster memory regions into bigger shared ones to be able to meet the limitation of MPUs having a small and fixed number of memory regions. For instance, our CheriFreeRTOS/OTA demo, which has more than 8 libraries compartments where each compartment has more than 8 memory resources (e.g., pointers), will not be able to be compartmentalised using MINION.

That is, unlike CompartOS, MINION will not be scalable the more the number of memory regions, compartments, and IO memory increase. Like ACES, MINION is still efficient for small embedded system applications. MINIONS is also a task-based software compartmentalisation approach as compartments are processes.

MINION is not OS-related, and it is not as advanced as CompartOS. We think of it as a more performance-optimised system for MPU-based domain switching that does not require costly OS checks performed during system calls. That makes it a bit similar to our CheriFreeRTOS’ **PMP-N-OBJS** or **PMP-N-LIBS** variants if we regard a CompartOS trampoline as a lightweight domain switcher. Still, CompartOS trampolines are semantically different (i.e., capability-based, linkage-based domain switching) and more performant as they do not incur a hardware cost of performing a system call or reconfiguring the hardware.

6.3 Mbed uVisor Hypervisor

uVisor [25] (deprecated and superseded by Arm’s Secure Partition Manager (SPM)) is a hypervisor developed by Arm to provide compartmentalisation for embedded systems using off-the-shelf MPUs. The architecture aims to enforce the principle of least privilege by only giving each protection domain access to the memory regions it needs.

The protection domain in uVisor is called a *box* which a thread or a process can execute within. The resources that boxes can isolate are stacks, code, data, and peripherals. To define what privileges each box entails, uVisor keeps an ACL list for each box. On every domain switch, the MPU is reconfigured according to each box’s ACL. Dynamic memory allocation is done through uVisor at large page granularity. The default page size is configured to be *SRAM_SIZE* / 16.

uVisor is similar to MINION as it can run an RTOS and other compartments in an unprivileged ring. However, it is different from MINION as it dynamically loads and creates compartments at runtime and is being able to place different software entities in a compartment such as interrupt handlers or linkage-based modules, besides threads. Thus, it is not only a task-based compartmentalisation approach. This makes it quite similar to our CheriFreeRTOS' **PMP-N-OBJS**, **PMP-N-LIBS**, and **PMP-4-TASKS** software variants, while the security policy is fed at build-time and enforced at (boot) runtime, as in CompartOS. That said, uVisor has stronger security requirements and enforcements, especially on dynamic memory allocation and temporal safety. Though not implemented or evaluated as they are not part of our model or requirements, CompartOS and its implementations could further adopt some of the uVisor ideas that include secure (or unprivileged) interrupt handlers and (heap) temporal safety, as discussed in Section 4.5. The main advantage for CompartOS over uVisor is that, similar to all MPU-based approaches, uVisor does not scale with the increasing number of resources and compartments. Further, uVisor does not aim to meet compatibility as a requirement. It is specific to Mbed as an EOS, and existing third-party applications and libraries (i.e., that are not originally written for Mbed or uVisor) need to be manually (re)written with Mbed and uVisor APIs to benefit from the software compartmentalisation advantage. Finally, uVisor does not try to provide fault handlers part of its model to improve availability, but it leaves that to the EOS.

6.4 TrustLite: A Security Architecture for Tiny Embedded Devices

TrustLite [27] is an enhanced MPU-based hardware security architecture for deeply embedded systems. TrustLite introduces a new enhanced version on an MPU called EA-MPU: Execute Aware MPU. EA-MPU attaches data memory regions to code regions.

The protection domain in TrustLite is called a *trustlet* which is simply a code region attached to it some data or peripheral memory regions, configured and isolated via the EA-MPU. On faults, the CPU could identify the currently executing code region from the faulting address, along with its associated data regions. This allows finer-grained compartmentalisation as different code regions could hold different privileges to the same resource at the same time without having to reconfigure the MPU. Further, unlike task-based compartmentalisation, modules within the same task can be isolated.

TrustLite further extends the CPU exception engine to allow secure execution of ISRs that do not necessarily trust the underlying OS.

TrustLite relies on a global ACL matrix that holds all of the trustlets. Like all MPU-based systems, this suffers from hardware scalability issues limiting the number of allowed memory regions and thus puts restrictions on the number of the overall trustlets that can be isolated (32 regions).

6.5 TyTan: Tiny Trust Anchor for Tiny Devices

TyTan [28] is a software security architecture based on Intel's Siskiyou Peak platform that has EA-MPU [27]. It targets tiny embedded devices and provides dynamically configurable compartments that can be loaded at runtime. Compartmentalisation in TyTan is manual and task-based; each task represents a protection domain, and context switches reconfigure the MPU to maintain isolation. The main contribution of TyTan is the ability to dynamically load applications and perform software attestation while meeting real-time requirements. EA-MPU can further be used to also secure tasks from the OS and also compartmentalise interrupt handlers.

Being an MPU-based software isolation approach, TyTan suffers from scalability issues. That is why the authors of the paper make it clear that it is aimed at tiny embedded and low-end devices. That is, TyTan is not unsuitable for more high-end embedded applications with rich software features and resources. Further, TyTan is customised to run on FreeRTOS and Intel's Siskiyou Peak. Thus, it is an incompatible generic security architecture, for example, when applying it to existing applications and operating systems.

6.6 CheriABI: CHERI Software Deployment in UNIX

CHERI has been under research in UNIX-based environments with MMU, prototyped in the CheriBSD OS (a CHERI-enabled fork of the FreeBSD OS). CheriABI [33] is an application-level software compartmentalisation technique in CheriBSD. The main software application in CheriABI is C/C++ language pointer safety at the user level with a few modifications to the FreeBSD kernel. Two compilation modes are supported for CHERI: hybrid and pure capability modes. In hybrid mode, pointers are integers as usual, and only those annotated with *__capability* keywords are protected by CHERI. CheriABI falls in the pure capability category where user processes are compiled to have all pointers, system call arguments and allocated C objects (such as malloc and TLS) represented as CHERI capabilities. This significantly enhances spatial memory safety in UNIX while it is still being compatible with native UNIX processes that are not aware of CHERI.

There is still ongoing research to have the FreeBSD kernel itself making full use of CHERI to compartmentalise the kernel components and enforce pointer safety. This is known as a pure capability kernel.

6.7 CheriRTOS: A Capability Model for Embedded Devices

CheriRTOS [35] is an early exploration of CHERI in embedded systems. Unlike CheriABI (that targets UNIX-based systems), CheriRTOS targets microcontrollers to offer hardware protection using 64-bit compressed CHERI-MIPS capabilities. The paper shows off how fine-grained memory protection, task isolation, secure heap management, and secure cross-domain transition can be implemented on CHERI.

The authors also argue that the MPU, which is usually being used for memory protection in safety-critical embedded systems, is impractical as it does not meet the fine-grained memory protection requirements. Configuring the MPU takes a considerable number of cycles in kernel mode and is inefficient when it comes to the power consumption and die area. The evaluation section shows that CHERI has potential in the embedded systems domain, and it needs more research to explore its application in this area.

Initially, I studied CheriRTOS and considered building on CheriRTOS as it is the first CHERI-based work in the embedded space. However, CheriRTOS did not meet all the requirements that I discuss below, while CompartOS could meet all of CheriRTOS' requirements. That is why I worked on CompartOS and CheriFreeRTOS; both are quite different (e.g., in implementation and design) and have significant advantages over CheriRTOS in areas such as stronger requirements, compartmentalisation technique, evaluation, and real-world software deployment.

This made it infeasible to have a side-by-side evaluation comparison between CheriRTOS and CheriFreeRTOS.

6.7.1 Compartmentalisation Technique

Unlike CompartOS (being an automatic linkage-based approach), CheriRTOS is strictly a manual, application-level, task-based software compartmentalisation approach. It does not have compatibility as a requirement. Thus, CheriRTOS is able to isolate user tasks from one another, but not other software subsystems such as unprivileged interrupt handlers, OS libraries, or linkage modules that can be put in a compartment, like in CompartOS and uVisor, without having to be encapsulated in tasks. For example, CheriRTOS cannot automatically compartmentalise large and mainstream embedded applications, such as the OTA, ECU, or TCP/IP demonstrators that we evaluated on top of CheriFreeRTOS, given that they are mostly (third-party) library-oriented rather than task-based. Further, CheriRTOS cannot compartmentalise EOS' subsystems such as device drivers and interrupt/fault handlers by placing them into linkage-module compartments like we did in TCP/IP demo in CheriFreeRTOS (see Section 4.4.4).

6.7.2 Memory Safety

CheriRTOS does not provide complete pointer safety as it does not build C/C++ applications in CHERI's PURECAP mode, but only in hybrid mode. It only focuses on CHERI-based domain switching and secure heap allocations. It uses *PCC* and *DDC* to only protect each thread's code and data sections, and further manually defined and added capabilities for inter-compartment calls and heap allocations. Building in CHERI's hybrid mode does not offer complete intra-compartment pointer safety like in the CheriFreeRTOS' PURECAP and CompartOS variants, and cannot catch (or recover from) similar intra-compartment TCP/IP vulnerabilities that we evaluated in Section 5.12.3. However, CheriRTOS does provide enhanced secure heap allocations that use the CHERI's sealing mechanism, which **PURECAP** does not offer nor implement. Further, CheriRTOS uses CHERI's sealing mechanism and hardware instructions to perform fast user-space domain crossing between tasks besides normal message-passing IPCs using buffers. Though quite different from CheriRTOS, CompartOS, by design, inherently supports fast domain crossing using trampolines as described in Chapter 4 and evaluated in Chapter 5.

That said, even though CheriRTOS is not evaluating CHERI in PURECAP mode, it is similar to CheriFreeRTOS' **PURECAP** variant that we implement and evaluate. As we discussed in Chapter 5, we regard **PURECAP** as our CHERI-based baseline when evaluating CompartOS in CheriFreeRTOS. The main similarities between CheriRTOS and **PURECAP** are that both are task-based, manual, and they context-switch *PCC* and *DDC* during task switches.

6.7.3 Availability and Fault Handling

CheriRTOS only attempts to provide inter-compartment, task-based isolation and to maintain spatial memory-safety (i.e., integrity and confidentiality) between compartments, but not availability or temporal memory-safety. Thus, it deploys CHERI as a mechanism for catching spatial memory-safety security violations, but it does not specify or provide a fault handling model or extension like in CompartOS. Availability is one of the most important requirements we have in CompartOS, and it is a significant advantage over CheriRTOS.

6.7.4 Evaluation

Unlike CheriRTOS, we implement and evaluate CompartOS against MPU-based software compartmentalisation techniques, at scale. We also further evaluate **PURECAP** and **CompartOS** in real-world, large, and complex mainstream applications. A brief evaluation comparison between CheriRTOS and ours (see Chapter 5) is as follows:

Scalability Even though CheriRTOS has scalability as a requirement, the paper does not evaluate it. For example, it does not have benchmarks or use-cases that have a large number of compartments or memory regions (as resources). In contrast, we evaluated scalability across a range of macro and micro benchmarks by increasing the number of compartments (e.g., by evaluating the performance of object modules in COMPARTOS-OBJS, besides COMPARTOS-LIBS), resources (where every pointer is a resource), and performance (by increasing the number of domain switches and IPCs).

Security CheriRTOS has security as a requirement, but it does not provide an empirical security evaluation as we did in Section 5.12.3. For example, it does not reproduce or demonstrate an inter-compartment security violation in a real-world application.

MPU Evaluation CheriRTOS briefly provides an approximate hardware evaluation by comparing the MPU versus CHERI hardware area overheads. However, it does not try to implement or evaluate MPU-based software protection models as we did in this dissertation.

Benchmarks CheriRTOS has only performed evaluation across performance benchmarks and not security-related or deployed use cases. The CheriRTOS benchmarks are mostly microbenchmarks (to evaluate critical code paths) and CPU benchmarks in MiBench. There are no clear compartmentalisation or memory protection benefits in such benchmarks, with logical security boundaries. In this dissertation, we have evaluated microbenchmarks and CPU benchmarks (e.g., CoreMark) similar to CheriRTOS. Additionally, as a significant evaluation difference, we actually compartmentalised multiple real-world, deployed, mainstream use cases to evaluate realism, performance, compatibility, security, and availability.

6.8 Summary

In this chapter, we described the most relevant state-of-the-art systems and how they compare against CompartOS and CheriFreeRTOS. We listed the limitations of most of the MPU-based systems such as ACES, TyTan, MINION, and uVisor. We argued that MPUs suffer from inherent scalability issues and are likely not suitable for high-end embedded systems that have many compartments and resources. We then discussed preliminary work that aims to secure a real-time embedded operating system using CHERI, which is called CheriRTOS. We compared CompartOS against CheriRTOS and listed significant differences in the models (e.g., task-based versus linkage-based), requirements, design, implementations, and evaluation.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This dissertation proposed CompartOS as a new security and compartmentalisation model for mainstream and large embedded (operating) systems. The hypothesis this dissertation tries to evaluate is that *CHERI and CompartOS can achieve greater compatibility, security, availability, and scalability than MPU-based and task-based protection for embedded OS compartmentalisation while they only add lightweight performance and development overheads when applied to secure unprotected mainstream embedded systems*.

Building on CHERI, CompartOS defends against most memory safety vulnerabilities—the most exploited type of vulnerabilities in software systems. CompartOS applies linkage-based software compartmentalisation as a technique to limit the effects of potential exploits due to software vulnerabilities only to the affected compartment.

In Chapter 3, we have defined the requirements for embedded systems and the gap that CompartOS is trying to fill there. We have also described the main design goals of CompartOS in the embedded systems field: compatibility, availability, security, and scalability. We described how the CompartOS model is versatile and largely adoptable in most mainstream embedded software systems as they all share common attributes. Further, we discuss a few fault handling techniques, part of the model, that help with partial recovery, which is an important aspect of maintaining availability in safety-critical embedded systems.

To be able to evaluate CompartOS in real-world applications, we implemented the CompartOS model in FreeRTOS, which is one of the most widely used embedded operating systems—thus the most representable of the systems we are targetting. Called CheriFreeRTOS, we described in Chapter 4 the implementation details to support CompartOS' linkage-based software compartmentalisation and how the secure dynamic loader integrates with the existing FreeRTOS threading implementation.

In Chapter 5, we have done a significant evaluation of CheriFreeRTOS as an application of the CompartOS model and compared it against unprotected MPU-based systems. First, we measured the hardware overhead to implement CHERI on top of an unprotected RISC-V processor; we showed that CHERI incurred 10% and 1.3% area and register size overheads, compared to 7.6% and 2% for MPU. We argue that this low overhead for the security benefits that CHERI and CompartOS offer. We then showed the efforts of porting FreeRTOS to CHERI and CompartOS in terms of lines of code. CompartOS and CHERI both took less than 400 LoC additions and changes, mostly in assembly. On the other hand, the MPU implementation in FreeRTOS took 2238 LoC. This proves how lightweight supporting CompartOS is in mainstream EOSes. We then evaluated the compatibility goal of CompartOS by compartmentalising embedded libraries and applications on top of FreeRTOS, written in over 100 KLoC. We chose already deployed use cases to also prove the realism and applicability of CompartOS. The use cases include a TCP/IP stack, servers, an OTA demo by Amazon, and a deployed safety-critical automotive use case by DARPA. It took very minimal effort to compartmentalise such applications by only building distrusting subsystems as linkage modules (e.g., libraries). We also evaluated the performance of CheriFreeRTOS across a range of micro and macro benchmarks.

We demonstrated that CompartOS adds a lightweight performance overhead to an unprotected baseline software variant, while it outperformed other MPU-based variants by a non-trivial margin. Finally, we reproduced real-world attacks based on published vulnerabilities in the FreeRTOS TCP/IP stack. CHERI could catch 10 out of 13 vulnerabilities, while CompartOS could recover from them. We evaluated fault handling and recovery at the end of the evaluation chapter within a safety-critical automotive demo; we showed the flexibility of CompartOS' fault handling while being able to meet real-time and safety-critical requirements there.

We hope the efforts spent in this thesis push the deployment of CHERI and CompartOS forward in embedded systems research and industry, as previous CHERI research did for general-purpose systems.

7.1 Future Work

As the CompartOS model (see Chapter 3) and the CheriFreeRTOS implementation (see Chapter 4) specify certain targetted systems, requirements, decisions, and a specific threat model, other embedded systems might have further requirements that CompartOS does not currently offer. However, we believe that the model and the implementation do not have restrictions and are extensible enough to support and implement further enhanced features beyond our requirements. In this section, we discuss a few approaches that can be adopted to meet other requirements, mature the implementation, and improve evaluation.

7.1.1 Applying the CompartOS model to Other Embedded Systems

Future work can aim to leverage the CompartOS model in the embedded systems field. We have only evaluated a prototype by implementing it in FreeRTOS on CHERI-RISC-V. We also have an initial work-in-progress prototype of applying the CompartOS model in RTEMS, which could further prove the model's applicability. Further prototypes and EOSes can also be implemented and evaluated on other processor architectures rather than just CHERI-RISC-V.

7.1.2 Security Trade-offs

We mainly focused on evaluating the compatibility, availability, and scalability of CompartOS and how it could recover from faults. While we have provided a CVE-based security evaluation, further sound security evaluation could be performed. This could be in the form of formally verifying a secure dynamic loader along with formalisation and evaluation of custom security policies. As we discussed in Section 5.8, we could also provide a security evaluation by introducing an external tracing tool or software that measures the set of capabilities each compartment has and compare that against the expected initial capabilities. On the implementation side, other CHERI and non-CHERI mechanisms can be deployed to catch security violations. As discussed in Chapter 3, we do not aim to catch specific security faults part of our model but to limit the effects of faults and recover from them in isolation. For instance, systems that require temporal safety may apply some solution (see Section 4.5), such as attaching multiple stacks for each compartment within a single thread. Some of the ideas in uVisor and CheriRTOS can also be applied to further enhance the security of heap management.

We have also only evaluated CheriFreeRTOS in CHERI's pure-capability mode. This provides complete spatial memory safety at a lightweight performance cost. For systems that do not require that level of fine-grained security and complete pointer-safety, CompartOS can be implemented in CHERI's compatible mode (see Section 2.11.1). This could be similar to what CheriRTOS did by sandboxing compartments only using PCC and DDC (or a few general-purpose CHERI capability registers in the captable), thus only protecting specific compartment sections at a coarser granularity.

7.1.3 Custom Security Policies

We have only relied on the default semantics of the ELF linkage model and C programming language as a default security policy in CheriFreeRTOS. Some tools and abstract specification languages can be developed to support custom security policy or access control mechanisms, similar to seL4's CAmkES [132]. This can further specify a direct-weighted-graph relationship between compartments, where weight represents a set of capabilities and permissions for communication APIs. This could enforce different subsets of a compartment's API a compartment can access, but not others. Weighted edges can be fine-grained by granting different permissions of the API to multiple compartments. For instance, one compartment could have read permission for an external variable in another compartment, while a second compartment could have read and write permissions.

7.1.4 Mapping CompartOS to Safety Standards and Certifications

On the application use cases, some safety-critical standards such as ARINC 653 could greatly fit with the CompartOS model. Most ARINC 653 implementations, however, are proprietary and closed source. Specific examples and approaches of that are discussed in Chapter 3.

7.1.5 Different Linker and Loader Implementations

The secure dynamic loader can be made more generic and reusable as a shared, secure, CHERI compartment bootloader for embedded systems similar to (or integrated with) u-boot [133] or SPM [134]. That said, complete static systems (i.e., that do not require threading, dynamic memory, partitioned software updates, allocation or compartment loads/unloads) may not need a dynamic loader. Instead, static linkers can be modified to take a static security policy as an input along with the linkage modules. Static linkers can further allocate partitioned capability tables (e.g., as ELF sections) for each linkage module and specify the communication bridges and capabilities properties in a relocation section. At boot time, a privileged initialisation function can populate capability tables based on the relocations. This could significantly reduce memory footprints, code size, and boot time spent in compartmentalising the system, but at flexibility cost.

7.1.6 System-level Compartmentalisation for General-Purpose OSes

This thesis has only focused on linkage-based software compartmentalisation at the system and application level for embedded systems. That said, we think that CHERI-based, linkage-based compartmentalisation can further be used in general-purpose systems. Previous CHERI work [32] has already shown how to sandbox application-level UNIX libraries using CHERI in a linkage-based approach. CompartOS' system-level, linkage-based compartmentalisation, or similar, can further be adopted in UNIX kernels themselves to compartmentalise their subsystems. For example, Linux device drivers (built as third-party linkage modules and not necessarily multi-threaded) can be compartmentalised in a single-address-space kernel when being dynamically loaded and linked against the kernel by calling *insmod* or *modprobe* Linux commands.

REFERENCES

[1] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. *Proceedings of the 23rd USENIX Security Symposium*, pages 95–110, 2014.

[2] Ang Cui and Salvatore J Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 97–106, 2010.

[3] David M. Nicoul. Hacking the lights out. Computer viruses have taken out hardened industrial control systems. The electrical power grid may be next. *Scientific American*, 305(1):70–75, 2011.

[4] V. Ananda Kumar, Krishan K. Pandey, and Devendra Kumar Punia. Cyber security threats in the power sector: Need for a domain specific regulatory framework in India. *Energy Policy*, 65:126–133, 2014.

[5] Molugu Surya Virat, S. M. Bindu, B. Aishwarya, B. N. Dhanush, and Maniunath R. Kounte. Security and Privacy Challenges in Internet of Things. *Proceedings of the 2nd International Conference on Trends in Electronics and Informatics, ICOEI 2018*, pages 454–460, 2018.

[6] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Others. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

[7] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. *Proceedings of the 20th USENIX Security Symposium*, pages 77–92, 2011.

[8] Charlie Miller and Chris Valasek. A Survey of Remote Automotive Attack Surfaces. *Defcon 22*, pages 1–90, 2014.

[9] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. *2015 13th Annual Conference on Privacy, Security and Trust, PST 2015*, pages 145–152, 2015.

[10] Sri Parameswaran and Tilman Wolf. Embedded systems security—an overview. *Design Automation for Embedded Systems*, 12(3):173–183, 2008.

[11] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760, 2004.

[12] Wikipedia contributors. Arm architecture - security extensions — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ARM_architecture#Security_extensions, 2022. [Online; accessed 16-February-2022].

[13] Wikipedia contributors. Arm cortex-m — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ARM_Cortex-M&oldid=1070148237, 2022. [Online; accessed 16-February-2022].

[14] Wikipedia contributors. Arm cortex-a — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ARM_Cortex-A&oldid=1061921480, 2021. [Online; accessed 16-February-2022].

[15] AWS Reference Integrations. https://www.freertos.org/aws-reference-integrations.html.

[16] Deos: Safety Critical RTOS for Avionics Applications requiring DO178C/ED-12C DAL A verification. https://web.archive.org/save/https://www.ddci.com/products_deos_do_178c_arinc_653.

[17] VXWORKS CERT PLATFORM PRODUCT OVERVIEW. https://web.archive.org/web/20211124075819/https://www.windriver.com/resources/product-overviews/vxworks-cert-product-overview.

[18] NASA: Magnetosphere Multiscale (MMS) Mission. https://mms.gsfc.nasa.gov/.

[19] RTEMS on board NASA MMS scheduled for launch this Thursday! https://www.rtems.org/node/118.

[20] Wikipedia contributors. Nxp coldfire — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=NXP_ColdFire&oldid=1031125403, 2021. [Online; accessed 16-February-2022].

[21] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David Hely, and Stephane Di Vito. An in-depth study of mpu-based isolation techniques. *Journal of Hardware and Systems Security*, 3(4):365–381, 2019.

[22] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Good motive but bad design: Why ARM MPU has become an outcast in embedded systems. *arXiv preprint arXiv:1908.03638*, 2019.

[23] FreeRTOS-MPU: Memory Protection Unit (MPU) Support. https://web.archive.org/web/20211031125921/https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[24] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[25] The Arm Mbed uVisor. `https://github.com/ARMmbed/uvisor`.

[26] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. {ACES}: Automatic compartments for embedded systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 65–82, 2018.

[27] Patrick Koeberl, Steffen Schulz, Ahmad Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014.

[28] Ferdi NAND Brasser, Brahim El Mahjoub, Ahmad Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. *Proceedings - Design Automation Conference*, 2015-July, 2015.

[29] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`, Feb 2019.

[30] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[31] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in multics. *Communications of the ACM*, 11(5):306–312, 1968.

[32] Robert N M Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, and Others. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[33] Brooks Davis, Robert N M Watson, Alexander Richardson, Peter G Neumann, Simon W Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, and Others. 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*, pages 379–393, 2019.

[34] Lawrence Esswood. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, University of Cambridge, 2021.

[35] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G Neumann, Alexander Richardson, Simon W Moore, and Robert N M Watson. CheriRTOS: A Capability Model for Embedded Devices. In *Proceedings of the 36th IEEE International Conference on Computer Design (ICCD '18)*, pages 92–99. IEEE.

[36] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. (February), 2018.

[37] Programming without a Net: Embedded systems programming presents special challenges to engineers unfamiliar with that environment. https://queue.acm.org/detail.cfm?id=644264.

[38] HiFive1 Rev B. https://www.sifive.com/boards/hifive1-rev-b.

[39] RDM-K64F: Freedom Development Platform for Kinetis® K64, K63, and K24 MCUs. https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F.

[40] BeagleBone Black. https://beagleboard.org/black.

[41] Wikipedia contributors. Apple m1 — Wikipedia, the free encyclopedia, 2022. [Online; accessed 31-January-2022].

[42] Wikipedia contributors. Hifive unleashed — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=HiFive_Unleashed&oldid=1006119760, 2021. [Online; accessed 31-January-2022].

[43] Wikipedia contributors. Raspberry pi — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=1072126107, 2022. [Online; accessed 16-February-2022].

[44] Wikipedia contributors. Beagleboard — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=BeagleBoard&oldid=1053954096, 2021. [Online; accessed 16-February-2022].

[45] Wikipedia contributors. Qoriq — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=QorIQ&oldid=1057508237, 2021. [Online; accessed 31-January-2022].

[46] RTEMS Real Time Operating System (RTOS) (26th July 2021). https://web.archive.org/web/20210726094718/https://www.rtems.org/.

[47] FreeRTOS: Real-time operating system for microcontrollers. https://www.freertos.org.

[48] RTEMS LibBSD. https://github.com/RTEMS/rtems-libbsd.

[49] FreeRTOS: Porting the TLS library - AWS. https://docs.aws.amazon.com/freertos/latest/portingguide/afr-porting-tls.html.

[50] FreeRTOS Over-the-Air Updates - FreeRTOS. https://docs.aws.amazon.com/freertos/latest/userguide/freertos-ota-dev.html.

[51] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[52] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[53] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[54] Wikipedia contributors. Buffer overflow protection — Wikipedia, the free encyclopedia, 2022. [Online; accessed 29-January-2022].

[55] FreeRTOS - Stack Usage and Stack Overflow Checking. `https://www.freertos.org/Stacks-and-stack-overflow-checking.html`.

[56] RTEMS - Stack Bounds Checker. `https://docs.rtems.org/branches/master/c-user/stack_bounds_checker.html`.

[57] Gerardo Richarte et al. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[58] Armv8-A architecture: 2016 additions. `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-a-architecture-2016-additions`.

[59] Memory Tagging Extension: Enhancing memory safety through architecture. `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety`.

[60] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[61] Robert N M Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W Moore, Steven J Murdoch, Robert Norton, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-907, University of Cambridge, Computer Laboratory, apr 2017.

[62] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1031, 2015.

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

[64] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[65] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[66] Robert N M Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. *USENIX Security Symposium*, 46:2, 2010.

[67] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[68] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[69] D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE CORP BEDFORD MA, 1976.

[70] Fernando J Corbató and Victor A Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, 1965.

[71] Charles Reis and Steven D Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.

[72] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[73] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot–a technique for cheap recovery. *arXiv preprint cs/0406005*, 2004.

[74] Linker (computing). `https://en.wikipedia.org/wiki/Linker_(computing)`, Jan 2022.

[75] ld(1) - linux man page. `https://linux.die.net/man/1/ld`, Jan 2022.

[76] Lld - the llvm linker. `https://lld.llvm.org/`, Jan 2022.

[77] Executable and linkable format. `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`, Jan 2022.

[78] Dynamic loading. `https://en.wikipedia.org/wiki/Dynamic_loading`, Jan 2022.

[79] Over-the-air programming. `https://en.wikipedia.org/wiki/Over-the-air_programming`, Jan 2022.

[80] 9.7. Dynamic Loader - RTEMS User Manual 6.b528508 (2nd July 2021) documentation. `http://web.archive.org/web/20210702090729/https://docs.rtems.org/branches/master/user/exe/loader.html`.

[81] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[82] Norm Hardy. The Confused Deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[83] Henry M Levy. *Capability-based computer systems*. Digital Press, 1984.

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

[85] William B Ackerman and William W Plummer. An implementation of a multiprocessing computer system. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 1–5. ACM, 1967.

[86] Maurice Vincent Wilkes and Roger Michael Needham. The Cambridge CAP computer and its operating system. 1979.

[87] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[88] Jochen Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

[89] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and Others. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[90] L4 Runtime Environment (L4Re). `https://l4re.org/doc/`.

[91] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[92] What is Proved and What is Assumed — seL4. `https://web.archive.org/web/20220720085604/https://sel4.systems/Info/FAQ/proof.pml`.

[93] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[94] Wikipedia contributors. Discretionary access control — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Discretionary_access_control&oldid=1054585135`, 2021. [Online; accessed 16-February-2022].

[95] Wikipedia contributors. Mandatory access control — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Mandatory_access_control&oldid=1069100685`, 2022. [Online; accessed 16-February-2022].

[96] Mark S Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003.

[97] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *NDSS*, volume 10, pages 357–374, 2010.

[98] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript. *Google, Inc., Tech. Rep*, 2008.

[99] Jonathan Woodruff, Robert N M Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer*

Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on, pages 457–468. IEEE, 2014.

[100] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. Cheri concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019.

[101] Prototype CHERI-MIPS processor on FPGA. `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/#cheri-mips-fpga`.

[102] Robert NM Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W Moore, Steven J Murdoch, Peter G Neumann, and Jonathan Woodruff. Bluespec extensible risc implementation: Beri software reference. Technical report, University of Cambridge, Computer Laboratory, 2014.

[103] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[104] CheriBSD. `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html`.

[105] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[106] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[107] RISC-V FOR FPGA. `https://bluespec.com/products#portable`.

[108] CHERI-RISC-V. `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html`.

[109] Arm Morello Program . `https://www.arm.com/why-arm/architecture/cpu/morello`.

[110] Robert N M Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, and Others. CHERI C/C++ Programming Guide. Technical report, University of Cambridge, Computer Laboratory, 2020.

[111] Hakem Beitollahi and Geert Deconinck. Analyzing well-known countermeasures against distributed denial of service attacks. *Computer Communications*, 35(11):1312–1332, 2012.

[112] Ryan Heartfield, George Loukas, Sanja Budimir, Anatolij Bezemskij, Johnny RJ Fontaine, Avgoustinos Filippoupolitis, and Etienne Roesch. A taxonomy of cyber-physical threats and impact in the smart home. *Computers & Security*, 78:398–428, 2018.

[113] Paria Jokar, Hasen Nicanfar, and Victor CM Leung. Specification-based intrusion detection for home area networks in smart grids. In *2011 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 208–213. IEEE, 2011.

[114] Luigi Coppolino, Valerio D'Alessandro, Salvatore D'Antonio, Leonid Levy, and Luigi Romano. My smart home is under attack. In *2015 IEEE 18th International Conference on Computational Science and Engineering*, pages 145–151. IEEE, 2015.

[115] Nikos Komninos, Eleni Philippou, and Andreas Pitsillides. Survey in smart grid and smart home security: Issues, challenges and countermeasures. *IEEE Communications Surveys & Tutorials*, 16(4):1933–1954, 2014.

[116] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6, pages 10–5555, 2006.

[117] Signal (IPC). `https://en.wikipedia.org/wiki/Signal_(IPC)`.

[118] Segmentation fault. `https://en.wikipedia.org/wiki/Segmentation_fault`.

[119] ARINC 653. `https://en.wikipedia.org/wiki/ARINC_653`.

[120] Standard Performance Evaluation Corporation - Wikipedia. `https://en.wikipedia.org/wiki/Standard_Performance_Evaluation_Corporation`.

[121] Isolation Benchmark Suite. `https://web2.clarkson.edu/class/cs644/isolation/`.

[122] DARPA SSITH – Securing our critical systems. `https://www.youtube.com/watch?v=nFmaRKwBO3U`.

[123] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[124] FreeRTOS+TCP and FreeRTOS+FAT Examples. `https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/TCP_FAT_demo_projects.html#Free_TCPIP_FAT_examples`.

[125] NVD - Vulnerabilities. `https://nvd.nist.gov/vuln`.

[126] FreeRTOS TCP/IP Stack Vulnerabilities — Mobile Security Blog. `https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/`.

[127] ARMmbed/mbedtls: An open source, portable, easy to use, readable and flexible SSL library. `https://github.com/ARMmbed/mbedtls`.

[128] intel/tinycbor: Concise Binary Object Representation (CBOR) Library. `https://github.com/intel/tinycbor`.

[129] aws/ota-for-aws-iot-embedded-sdk. `https://github.com/aws/ota-for-aws-iot-embedded-sdk`.

[130] CAN bus - Wikipedia. `https://en.wikipedia.org/wiki/CAN_bus`.

[131] SAE J1939 - Wikipedia. `https://en.wikipedia.org/wiki/SAE_J1939`.

[132] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. 32(1):1–70.

[133] Wikipedia contributors. Das u-boot — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Das_U-Boot&oldid=1061095005`, 2021. [Online; accessed 1-February-2022].

[134] Secure Partition Manager (MM) – Trusted Firmware-A. `https://trustedfirmware-a.readthedocs.io/en/latest/components/secure-partition-manager-mm.html`.