MANCHESTER 1824
The University of Manchester

Exploring Software Compartmentalisation with Hardware Capabilities

A thesis submitted to the University of Manchester for the degree of
Master of Philosophy
in the Faculty of Science and Engineering

2023

John Alistair Kressel
Department of Computer Science

Contents

Word count: 20588

List of figures

List of tables

List of publications

The following publication is based on the work done as part of this MPhil.

J. A. Kressel, H. Lefeuvre, and P. Olivier, "Software Compartmentalization Trade-Offs with Hardware Capabilities," 12th Workshop on Programming Languages and Operating Systems (PLOS 2023).

Abstract

Compartmentalisation is a form of defensive software design in which an application is broken down into isolated but communicating compartments. Retrofitting compartmentalisation into existing applications is often thought to be expensive from the engineering effort and performance overhead points of view. ARM Morello combines a modern ARM processor with an implementation of Capability Hardware Enhanced RISC Instructions (CHERI) aiming to provide efficient and secure compartmentalisation by using CHERI capabilities to isolate portions code and data. CHERI provides a hybrid mode, where capabilities can be used alongside standard pointers in software. This promises to reduce the engineering burden associated with implementing compartmentalisation in legacy software by eliminating the need to port entire code bases.

This thesis explores possible compartmentalisation schemes available to developers in a single address space environment with CHERI in hybrid mode, and then proposes two approaches representing different trade-offs in terms of engineering effort, security, scalability, and performance impact. These approaches are described, implemented and evaluated on a prototype unikernel running bare metal on the Morello chip, compartmentalising two popular applications, SQLite and Libsodium. Unikernels feature no memory isolation between the kernel and application, with both occupying the same address space for performance reasons, which raises security concerns. CHERI compartmentalisation in hybrid mode is, therefore, explored as a way to establish isolation between components within a unikernel, with a potentially low engineering effort while preserving the performance advantages of sharing a single address space.

The evaluation shows that CHERI, in hybrid mode can achieve compartmentalisation within a single address space unikernel environment, at a performance overhead which is comparable to that achieved with Intel MPK and outperforms that achieved with Intel EPT. Furthermore, it shows that the isolation achieved, outperforms the user-kernel separation provided by Linux. However, the evaluation demonstrates that the engineering cost of applying CHERI compartmentalisation in hybrid mode using fine-grained capabilities for inter-compartment communication is high, making this approach impractical outside of small-scale scenarios. To tackle this issue an alternate data sharing method is proposed, which trades off scalability and security to reduce the engineering effort.

Lay abstract

Security is often an afterthought in the design of a software system. With the increasing ubiquity of computer systems in critical infrastructure, it is important to isolate components of a system from each other to reduce the damage which can be done in the event of a successful attack on any component. Compartmentalisation offers a way to do this, by splitting software into compartments which have only the minimum privileges needed to complete their task. Many technologies exist to do this, including CHERI, a hardware implemented technique which can be used to restrict the parts of a system accessible to compartments. Splitting software systems into compartments is often associated with a performance cost and a significant engineering challenge. This thesis presents the design, exploration and evaluation of various methods for implementing CHERI based compartmentalisation, which trade off performance, engineering effort and scalability for improved security. In contrast to much of the existing work using CHERI, which is evaluated using simulators or soft cores, the system is implemented and evaluated on Morello hardware, an extension to a standard 64-bit ARM processor implementing CHERI.

Declaration of originality

I hereby confirm that no portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright statement

Acknowledgements

I wish to thank my supervisor Dr. Pierre Olivier for providing me with invaluable support, guidance and mentorship in my research. I also wish to thank Professor Mikel Luján and Hugo Lefeuvre for their advice and support this year. Next, I would like to thank Abhinav Krishnan for his patient listening to my endless ramblings. Additionally, I would like to thank Dr. Andrew Attwood for his advice and feedback. Finally, I would like to extend my gratitude to my family and friends for being patient and supportive, thank you!

Chapter 1

Introduction

Software compartmentalisation is one of the ways to enforce the principle of least privilege [1]. Compartmentalisation enforces isolation between components of a software system, granting compartments only the minimal privileges they need to function. If a component of a compartmentalised system is subverted, the damage the attacker can do is limited to the privileges granted to the compromised compartment [2], [3]. Contrary to many other protection techniques, compartmentalisation allows defending against yet unknown and future vulnerabilities in existing code bases [4]. Many approaches have been proposed in recent years, utilising different hardware and software isolation mechanisms to compartmentalise libraries [5]–[15] as well as code at other granularities, including functions [16]–[19] and device drivers [20].

Morello [21] is an extension to the ARMv8-A architecture implementing the Capability Hardware Enhanced RISC Instructions (CHERI), designed specifically to enable high-performance and scalable compartmentalisation [4], [22], [23]. This is achieved by enforcing compartment bounds on most memory loads and stores in hardware, and letting communicating compartments securely grant memory access to each other using so-called hardware capabilities, a mechanism similar to fat pointers [24]–[26] implemented in hardware to restrict accesses to shared memory at a fine (byte-level) granularity. Compartmentalisation can be achieved by using CHERI capabilities to isolate the code and data accesses of compartments. To eliminate the need to port entire code bases to use capabilities, CHERI provides a hybrid mode, where regular pointers can be selectively replaced with capabilities by the developer. Additionally, all pointer memory accesses are implicitly checked against a set of global architectural capabilities. The promise of hybrid mode is, therefore, low engineering effort to integrate capabilities and compartmentalisation into an existing code base.

When retrofitting compartmentalisation to existing code bases, a key challenge is keeping refactoring costs low [27]. This is crucial not only for reducing the cost of deployment, but also to reduce the number of errors made during the compartmentalisation, which can undermine its efficiency or security guarantees [28]. Work exploring compartmentalisation with CHERI is so far limited mostly to solutions implemented in pure capability mode [4], [6], [29]–[31], where all pointers

are replaced with capabilities. This presents a need for an exploration of compartmentalisation in hybrid mode, especially given the promises of low engineering effort needed to integrate it into an existing code base. These existing works are further limited to MIPS/RISC-V emulated or FPGA prototypes, making it hard to understand the real-world performance one would observe on a hardware processor. In that context, the recent availability of Morello raises the following research questions which will be addressed in this thesis:

RQ1 Which compartment models are possible using Morello, using what programming abstractions, at which refactoring costs, and how do they scale?

RQ2 How does Morello's compartmentalisation performance compare to other single address space compartmentalisation mechanisms, such as Intel Memory Protection Keys (MPK)?

RQ3 What security properties does CHERI/Morello-based compartmentalisation offer, versus mechanisms such as MPK?

For this purpose, an existing compartmentalisation-oriented unikernel (libOS), FlexOS [15] is ported to run bare metal on Morello hardware in hybrid mode. Unikernels are statically linked and code and data resides in a single address space, which is done to improve performance. However, the lack of any isolation between application and kernel components is a security concern. Introducing compartmentalisation to restore isolation between certain parts of the system is desirable [18], [32]. Doing this in a way which maintains the performance of unikernels and does not require significant engineering effort is a requirement. In addition, exploring compartmentalisation within a single address space unikernel will allow findings to be applied to other single address space settings such as processes.

To achieve this, FlexOS is extended by developing compartmentalisation programming abstractions relying on CHERI hardware capabilities, each representing a particular trade-off in terms of porting costs, security guarantees, and their ability to be applied to many compartments. These include manual sandboxing as advocated by CHERI's designers [4], with every shared buffer protected by fine-grained capabilities and additionally an approach relying on a single region of shared memory between pairs of communicating compartments. These abstractions are used to compartmentalise popular open source software, SQLite [33] and Libsodium [34], at different isolation granularities: libraries and functions. The porting costs and degree of security of these solutions are evaluated. Further, their performance when executing on the Morello chip is analysed, comparing these results to that of other single address space isolation mechanisms, Intel MPK and Intel EPT, which have previously been used to introduce compartmentalisation to FlexOS.

The performance evaluation shows that the performance achieved by CHERI compartmentalisation in hybrid mode when applied to FlexOS using the shared memory approach is comparable to that achieved with Intel MPK and outperforms Intel EPT. Additionally, FlexOS with CHERI hybrid mode compartmentalisation applied still outperforms the same application running under Linux with a similar model of isolation.

Regarding the engineering cost of integrating compartmentalisation, while it is possible to make trade-offs to reduce the engineering cost, the challenge of sharing data in a single address space system means that hybrid mode using fine-grained capabilities presents engineering issues which require design compromises to overcome.

1.1 Thesis Structure

Chapter 2 of this thesis first takes an in-depth look at the motivation for compartmentalising software, as well as the concepts needed to implement it. Further, it investigates the various compartmentalisation technologies which have been applied in the literature to give context of where this work stands in the broader field and finally, describes in detail how CHERI works and how it has been applied to the problem of compartmentalisation.

Chapter 3 introduces the design of the proposed system from a high level, including how data can be shared between compartments and considers the performance and engineering costs, as well as the scalability of the proposed solutions.

Chapter 4 goes on to detail the steps taken to port FlexOS to the Morello platform and how the system is implemented, including the precise programming abstractions and annotations needed for the different compartment models.

Next, chapter 5 evaluates the implemented system with respect to the performance cost, the engineering effort needed to port software to use it and the security guarantees it provides. These are compared to other popular mechanisms: MPK and EPT.

Chapter 6 presents the conclusion of this thesis, followed by the contributions made by this thesis and a description of proposed future work.

Chapter 2

Background

The following chapter considers, first, the motivation for researching compartmentalisation and how it is an important technique, as well as defines the important terms and principles needed to understand compartmentalisation and the available compartment models. Next, different mechanisms which can be used to enable compartmentalisation are examined in detail, followed by an in-depth look at CHERI, which is needed to understand the work presented in the remainder of this thesis.

2.1 Introduction To Compartmentalisation

Compartmentalisation stems from the principle of least privilege, first defined by Saltzer *et al.* [1] in 1975, as "...the least set of privileges necessary to complete the job". In practice this means that system failures or malicious activities are limited to only part of the system and so the damage caused is also limited.

The study of software compartmentalisation focuses on the techniques and mechanisms to enable previously monolithic software systems such as applications to be partitioned into a set of smaller, isolated compartments with limited privileges, controlled data sharing and isolated private data. With the ubiquity of computer systems in everything from critical safety systems and the Internet of Things, to the infrastructure underpinning modern society, it becomes ever more necessary to protect these systems from malicious activities.

2.1.1 What Is A Compartment?

A compartment can be simply defined as a unit of isolation encompassing part of the code and data of a larger entity. Compartments are granted the minimum bounds and permissions which are needed to execute their intended task. In practice this means that a compartment may contain some code such as a function or a library as well as data which is intended to be kept private. Limited data sharing is then established between compartments to allow controlled access to external resources.

2.1.2 The Need For Compartmentalisation

The following section examines the real need for compartmentalisation in modern computer systems, starting with high level examples of well-known and costly security vulnerabilities, followed by a closer look at the types of attacks which can be thwarted by compartmentalisation when it has been applied properly.

It Affects Us All

Well known examples of security vulnerabilities which caused widespread disruption include: Heartbleed [35] which affected over 50% of all websites and allowed attackers to read private data from HTTPS encrypted websites by supplying an invalid length with a heartbeat message. A similarly critical and remotely exploitable vulnerability was discovered in log4j [36], the popular Java logging library. Both vulnerabilities resulted in widespread disruption and millions of vulnerable systems.

Both of these examples demonstrate the need for compartmentalisation to mitigate such memory safety issues. Such vulnerabilities are ever present in large code bases [37] and indeed, the number of new vulnerabilities reported is increasing every year. While problems can be fixed with patches and updates, these can be time-consuming to apply, and not all systems will receive fixes in a timely manner [35]. These points are critical, since during this time, vulnerabilities can be exploited.

When a vulnerability is exploited in an uncompartmentalised system, the exploited code or module has more privileges than it needs to execute its task, meaning the rest of the system is also liable to compromise. This presents attackers with the potential to steal sensitive data or perform other malicious tasks, regardless of how seemingly unimportant the compromised code was. Consequently, a system is only as secure as the weakest component in that system. Large and complex software systems may use hundreds of libraries.

How Compartmentalisation Can Help

Many different attack scenarios exist. Compartmentalisation research has focused on many of them. In addition to currently known attacks, compartmentalisation can be used to defend against yet unknown attacks by limiting the exploitable surface of a vulnerability. Compartmentalisation does this by reducing the privileges granted to a software component.

Memory safety refers to bugs and attacks which target unsafe memory accesses, for example buffer overflows and underflows. Such issues are particularly prevalent in C code bases because C is not a memory safe language, relying instead on

manual memory management by the developer, which is prone to bugs. Google and Microsoft have separately identified that 70% of their bugs stem from memory safety issues [38], [39]. Compartmentalisation can mitigate the damage caused by such bugs by limiting the accessible memory given to compartments. Much work has focused on memory safety [22], [40], [41]. Indeed, capabilities implemented in both hardware and software [24], [25] such as CHERI [22] specifically target memory safety vulnerabilities. Additionally, a growing area of interest for researchers is the use of memory safe languages in domains which have typically used C for the past decades, such as the inclusion of components written in Rust in the Linux kernel [42].

Supply chain attacks are possible because large code bases often use many different libraries and software components from different and untrusted sources. This is done to ease development by using tried and tested software libraries, for example cryptographic libraries. Vulnerabilities or malicious code present in such components can cause damage. Code can be acquired from many sources [43] including package managers. Compartmentalisation can be applied to separate application components into distrusted compartments, limiting the scope of such attacks [44], [45].

Side channel attacks gather additional information such as frequency and timing information to exploit systems [46]. Compartmentalisation can be used to protect against some side channel attacks by reducing the oversharing and leakage of data and enforcing security boundaries. A growing body of work is focused on mitigating side channel attacks [47]–[49], including attacks such as Spectre [50]–[53].

Fault tolerance is important in safety critical systems where continued operation despite malfunctioning or exploited code is necessary. For example, a denial-of-service attack on safety critical services, caused by exploiting vulnerable software or buggy device drivers, can result in a crash which can be very damaging and difficult to recover from. Consider a real-time safety system at a nuclear facility, or software controlling the movements of an autonomous vehicle; a crash in such systems could lead to catastrophic consequences. Protection against attacks and faults causing crashes can be achieved using compartmentalisation, since code is unprivileged and limited in its scope for damage and disruption [30], [54].

Protecting secret data such as cryptographic secrets and other highly sensitive data can be achieved through compartmentalisation. By isolating secret data and code which manipulates it, the data cannot be stolen via attacks on other components, for example the heartbeat component of OpenSSL [55] which was exploited by Heartbleed. A large body of work has focused on protecting private information [16], [56]–[60].

2.1.3 What Makes Good Compartmentalisation?

So far the need for compartmentalisation has been examined. Next, the criteria for a good implementation are detailed. These criteria are critical for producing compartmentalisation which can be adopted by real software and thus have a tangible impact.

Low Performance Overhead

Partitioning monolithic applications into smaller components running within their own isolated compartments can potentially add a significant performance overhead to an application. This stems from a number of factors. Firstly, compartments can often no longer communicate as before by de-referencing pointer arguments. Instead, communication is frequently done via shared memory regions, inter-process communication, or using other mechanisms which are costlier due to the additional data copying which is required. Many isolation mechanisms also rely on additional checks inserted via instrumentation or traps, to check and catch an illegal code or data accesses [10], [61], [62]. Switches between compartments are also often inserted in the form of call gates in place of traditional function calls, which themselves can add significant overhead, since gates can be used to perform tasks such as switching compartment stacks and address space switching [15].

Security can come at a cost. The challenge to design a compartmentalisation scheme is to balance the performance and security needs to an acceptable degree to ensure that any performance cost is minimised, while still offering improved security. Systems with high performance overhead will struggle to find adoption in real applications, since lower performance will mean a greater hardware and energy cost to deploy large applications. DARPA CPM defined an acceptable CPU overhead of <15% for OS level isolation and <5% for application level isolation, in their call for proposals [27].

Low Engineering Effort

Alongside performance considerations are implementation costs, or the engineering effort which is required to port or retrofit an application to take advantage of compartmentalisation technologies and primitives. Legacy applications are often not designed with compartmentalisation in mind [28], resulting in complex data dependencies which can be challenging to unpick and isolate from an entangled system. Consequently, any solutions which require too much effort from a developer, are unfeasible for a number of reasons. Firstly, code may be decades old and either poorly maintained, poorly documented or both. Additionally, with increased complexity comes an increased risk of human error, which leads to poor compartmentalisation practices. Trying to retrofit compartmentalisation into such

code bases is costly in terms of both time and effort and can be a significant barrier to adoption.

Flexible Compartmentalisation Granularity

Flexibility when implementing compartmentalisation is important to reduce both the performance overhead and engineering effort involved. By enabling fine-grained compartmentalisation, a developer can choose to isolate large, coarse components such as libraries, smaller units of code such as functions, or just a few critical data structures such as cryptographic keys, depending on their performance and engineering budget, as well as the threat model. Some isolation mechanisms such as page table based approaches only allow coarse compartmentalisation down to the granularity of a page, which can be impractical for small and light implementations due to hardware pressures and inefficient use of memory pages, if only small quantities of data are shared.

Scalability

Scalability is the ability to create a few or many compartments without the cost of adding more becoming a burden. This is important because a solution which does not scale well will force potentially unwanted compromises, for example the inability to partition an application into the required number of compartments could lead to weakened security.

Small Trusted Computing Base

The Trusted Computing Base (TCB) is the hardware and software of a system which is trusted to be error free and correct in order for security properties and assumptions to hold true. If a flaw or bug exists in the TCB, it can have wide-ranging and potentially devastating consequences. Since the TCB must be error free, it is desirable for this portion of the system to be as small and simple as possible, to enable easier verification and also reduce the possibility for errors.

Secure Compartment Interfaces

Lefeuvre *et al.* [28] defined compartment interface vulnerabilities (CIVs) and established that simply compartmentalising an application is in most cases not enough. Trust boundaries are introduced where previously there was mutual trust. This means that compartments can be made to misbehave or share private information simply because the exposed interfaces do not sanitise malicious data which is passed into them, for example through confused deputy attacks [63]. Therefore,

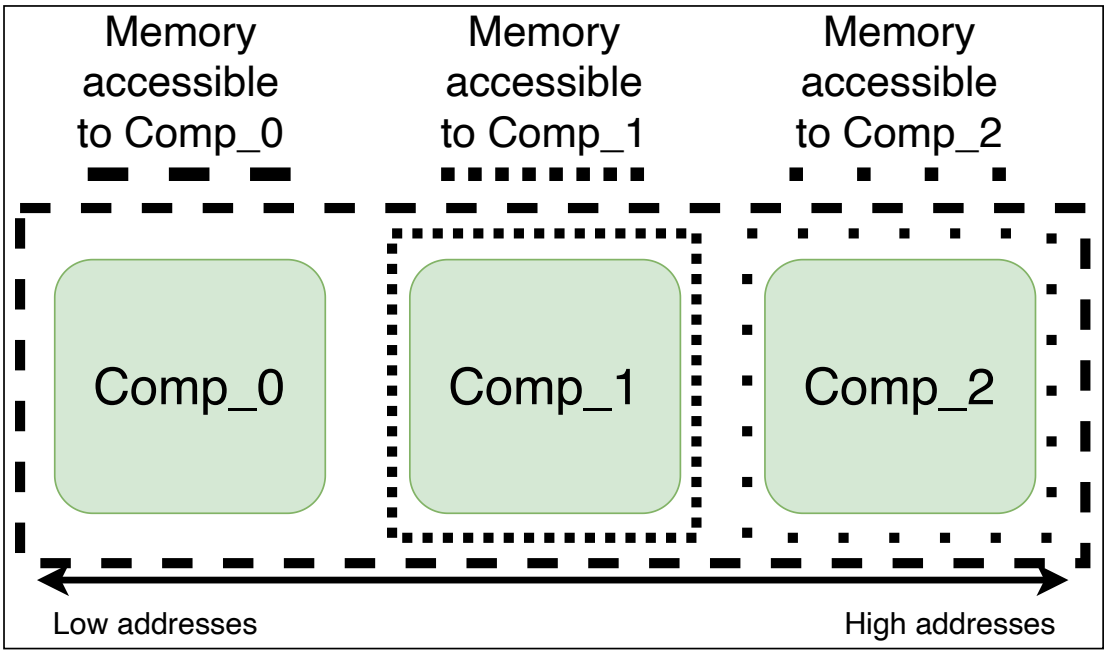

Figure 2.1. Example of sandboxing. Compartments 1 and 2 are sandboxed.

hand in hand with compartmentalisation must be a re-evaluation and hardening of the trust boundaries which this exposes.

2.1.4 Trust Models

The different trust models which are available to someone wishing to compartmentalise a system are described in the following sections.

Sandboxing

Sandboxing can be thought of as a sandbox a child plays in. It is a place for them to explore and play, but they are also contained within to avoid them being able to cause damage to anything outside of the sandbox. In a computing sense, an untrusted code module is placed into a sandbox to prevent it from accessing the rest of the system. It is given only what it needs to function and is self-contained within it. In this trust model, the contents of the sandbox are untrusted and everything outside the sandbox is kept safe and implicitly trusted. Many popular applications and libraries such as Chromium [64] and OpenSSH [3] use sandboxing to contain potentially unsafe execution of components such as the decompression and decoders, which are frequently the source of exploits. Sandboxing is illustrated in Figure 2.1.

Safeboxing

In contrast to sandboxing, safeboxing trusts what is inside the box and protects it from what is outside. This means that access to data within a safebox must be

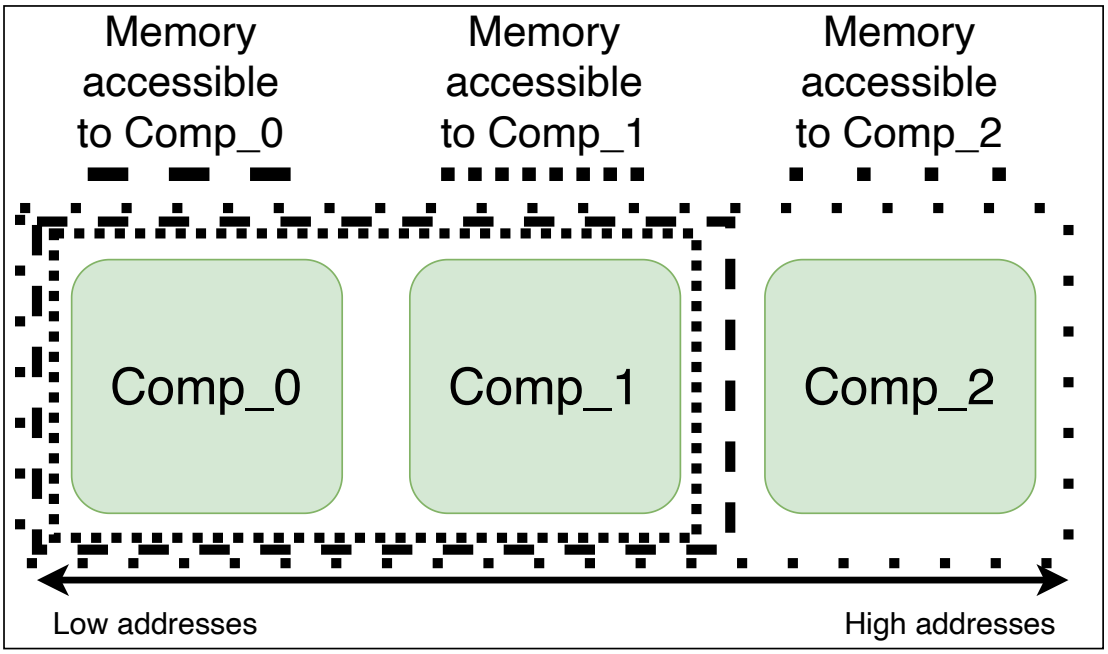

Figure 2.2. Example of safeboxing. Compartment 2 is safeboxed.

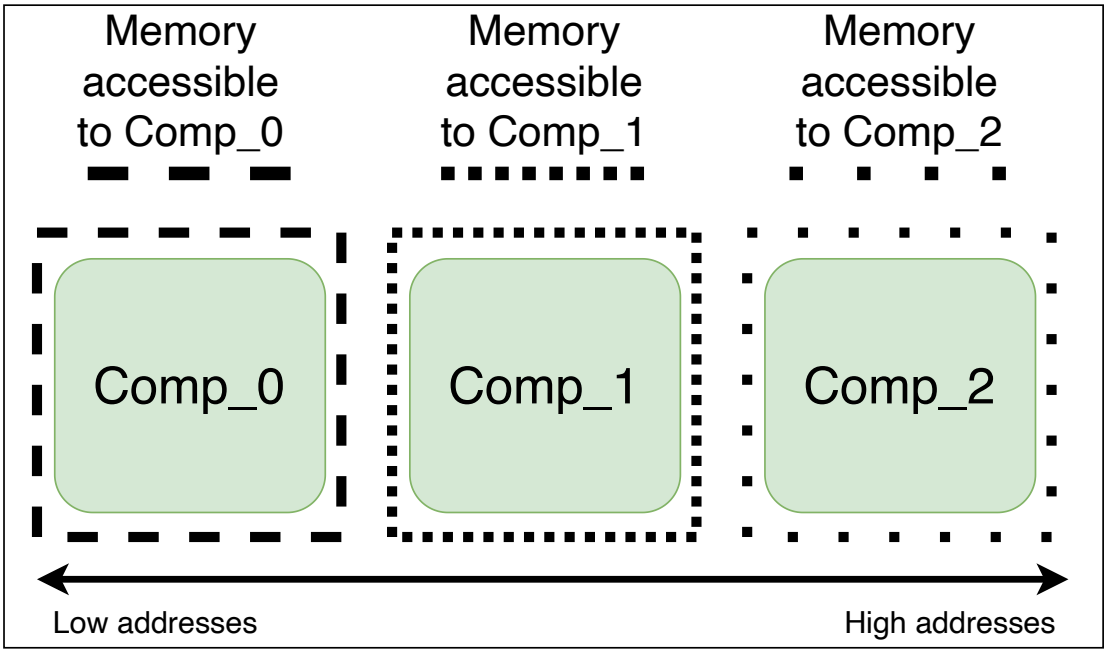

Figure 2.3. Example of mutual distrust.

carefully controlled and data entering from outside the safebox must be carefully sanitised. An example of a use case for safeboxing could be a cryptographic function which is trusted and is protected from compromise by preventing the rest of the system from accessing it. Safeboxing is illustrated in Figure 2.2.

Mutual Distrust

Mutual distrust is a middle ground between safeboxing and sandboxing. Here, entities distrust each other. Private compartment data must be kept isolated from all other compartments. This model has been used by works including FlexOS [15]. Mutual distrust can be used between an application and its libraries,

where each would like to protect itself from the other in case of compromise in another compartment. Mutual distrust is illustrated in Figure 2.3.

2.2 Mechanisms

Various techniques have been implemented and explored to enforce isolation, these are examined in the following sections.

2.2.1 Page Tables

Page tables are used by the hardware memory management unit to translate virtual to physical addresses. They are widely implemented in modern processors. This gives rise to many approaches to page table based isolation, including Arbiter [65], which provides isolation between application threads by using a different page table for each and LwCs [66] which provides page table based isolation within a process. Page table based isolation is often combined with privilege separation, such as that between user and kernel.

The following sections explore common page table based approaches to isolation.

Processes

Processes are used to provide isolation between different executing code, by giving each a distinct virtual address space, through the use of a process specific page table. Access to system resources is controlled and granted by the kernel via system calls. A process can be isolated not only from other processes, but also from critical kernel components. Switching compartment involves the kernel switching out the page table for that of another process.

Many compartmentalisation works take advantage of processes to implement isolation between different components of a system. Cali [13] and CompARTist [67] use processes to isolate an application from its libraries, with isolated libraries running in their own process. PtrSplit [12], Privman [68], Privtrans [57] and Salus [69] split an application into two differently privileged compartments. These compartments then run in separate processes. Wedge [59] provides OS primitives to allow developers to split monolithic applications into process-isolated compartments. ProgramCutter [70] uses dynamic analysis to automatically partition an application into separate processes at function granularity.

Access Control Bits

Access control bits are implemented as bits within a page table entry [71], [72] and enable access controls to be enforced for different regions of memory, including

the setting of read, write and execute permissions. This is an important feature, since it can prevent a malicious actor from modifying the executable portion of an application in memory, thereby circumventing protection mechanisms. Additionally, it can prevent writable portions of memory from being executed, which could otherwise allow an attacker to execute arbitrary instructions in memory.

Second Level Address Translation

Second Level Address Translation (SLAT), also referred to as nested page tables in the literature [73] is a type of hardware assisted virtualisation which makes guest to host address translation more efficient. This means that code can easily be virtualised. SLAT has been implemented by multiple hardware vendors including Intel [74], AMD [75] and ARM [71].

SLAT can be used to compartmentalise an application at the function level such as with SeCage [76] and Virtines [17] all the way to a more coarse grained library level compartmentalisation such as has been implemented by FlexOS [15] with two compartments.

Virtual Machines

Virtual Machines were first deployed by IBM with the VM370 time-sharing system [77] and then formally defined by Popok and Goldberg in 1973 [78]. They were introduced as a means to run multiple operating systems or other software which requires a bare-metal view of the machine [79], on the same hardware. Since then, they have grown in popularity, and are primarily used by Infrastructure as a Service (IaaS) providers to run multiple operating systems on server hardware, to serve the needs of numerous different clients. A hypervisor is used to coordinate the various virtualised systems.

Whilst useful to isolate operating systems running on the same hardware from each other, the idea can be extended to isolating applications from each other, by running applications on small operating systems like unikernels (LibOS) which are lightweight. Indeed, works such as FlexOS [15] take advantage of virtualisation to run an application in a unikernel on a host system. NGSCB [80], Proxos [81] and Minibox [82] focus on OS level virtualisation to improve security. Overshadow [83], SP3 [84], Inktag [85] and virtual ghost [86] focus on isolating applications from an untrusted operating system.

Mondrian Memory Protection (MMP)

Witchel *et al.* proposed Mondrian Memory Protection [87], [88] as a way to augment traditional page table based memory management with a special permissions

table for permissions which can be applied to memory regions down to word granularity. This is supported by a *protection lookaside buffer (PLB)* [89] and *register sidecars*, which contain the permissions associated with the address in the address registers, used to improve performance.

This allows for multiple compartments within a single address space. Compartment switches are performed by calling addresses or return addresses which are marked in the permissions table as being cross domain calls. This then triggers a compartment switch.

Memory Protection Keys (MPK)

Memory protection keys use bits in a page table entry to associate a numeric key with entries. They target isolation within a singe address space. Upon attempting to access memory via a load or store instruction, the current key is checked against the key associated with the page table entry corresponding to the attempted memory access. If the two do not match, a page fault is triggered. Therefore, a protection key corresponds to a compartment. MPK has been implemented by a number of architectures, including x86 [72], [75], RISC-V [11], [90], IBM Power [91] and ARM (ARM domains) [92]. These implementations allow for between 16 (x86 and ARM) - 1024 (RISC-V) compartments to be implemented. MPK on RISC-V is not currently widely available in hardware, and support for ARM domains was dropped in AArch64 meaning that limited work has used the mechanism [58]. This leaves x86 MPK as the only widely proliferated and available implementation of MPK.

x86 MPK compartment switches can be done from user space, via the unprivileged `PKRU` register, which means that compartment switches can be carried out with low overhead without needing expensive syscalls or other mechanisms to obtain the correct switching privileges. The downside to unprivileged user space access is that x86 MPK on its own, is not sufficient for secure isolation and must be paired with other techniques [11] such as control flow integrity or binary scanning to ensure that a malicious actor does not arbitrarily switch compartments.

MPK mechanisms have been utilised in a number of compartmentalisation works in recent years, including Donky [11], FlexOS [15], Hodor [9], CubicleOS [14], Shreds [58], libMPK [93], Enclosure [44] and by Sung *et al.* using RustyHermit [18].

Limitations of Page Tables

Page table based isolation mechanisms often suffer from two key limitations, making them potentially problematic for compartmentalisation.

Translation Lookaside Buffer (TLB) Performance The TLB is implemented in hardware to cache virtual address translations, reducing the cost of performing these translations. Process context switches will frequently cause the TLB to be invalidated and flushed. For example, when switching processes, the page table will be switched out. Although certain modern architectures are compartment aware, it is still likely that the number of TLB misses will increase due to compartment switching. Refilling the TLB with useful predictions is expensive, since it involves accessing main memory. MMP may also suffer from a similar performance penalty to the PLB. Since MPK is used in-process, this does not apply.

Granularity Since page table based approaches are based on the page table, it follows that the smallest unit of isolation is a page. However, pages are coarse. They are typically sized at 4KB, although many modern processors use larger pages. Most data structures will be smaller than the size of a page and so memory usage will be inefficient to facilitate isolation, or a significant amount of data may be overshared.

2.2.2 Software Fault Isolation (SFI)

Software Fault Isolation, introduced by Wahbe *et al.* [94], is a runtime method for sandboxing memory accesses to occur only within defined bounds. This is done by instrumenting a binary with instructions which check the addresses being used, before they are used. Instrumentation can be inserted by the compiler at compile time, or retrofitted to a binary using binary rewriting. In order to avoid SFI instructions being skipped by an attacker, SFI is usually paired with some form of verification or control flow integrity [95], [96]. SFI normally applies to isolation within a single address space. A large body of research has looked at SFI, including ARMor [97], NaCl [61], [98], /CONFIDENTIAL [99], PittSFIeld [62], Rocksalt [100], XFI [101], LXFI [102], Datashield [103], Occulum [104] and RLBox [10]. A downside of such a technique being implemented is the performance penalty imposed by adding additional instructions to the application.

In recent years, WebAssembly [105], [106] has emerged as a way to implement SFI with lower overhead. WebAssembly is a memory safe binary instruction format which can be used to run code in sandboxes with high performance. This is useful in applications, such as browsers, to restrict unsafe and untrusted code execution. Additionally, it has been explored in numerous works as a mechanism for compartmentalisation [10], [107]–[110]. RLBox has used Wasm to implement sandboxing in the production FireFox browser [10], [111].

2.2.3 Safe Programming Languages

Popular memory safe programming languages such as Java [112] and Ruby [113] have existed for decades. Memory safe languages have traditionally relied on runtime checks and memory management, which impose a high performance penalty. This has typically made such memory safe languages unsuitable for low-level, performance sensitive systems software. KaffeOS [114], JX OS [115] and J-Kernel [116] implement OSes using Java, however, they are limited by performance and functionality. Singularity OS [117] uses Sing# to achieve memory safety. SPIN [118] uses language safety for security within the kernel only. MirageOS [119] uses OCaml [120] for memory safety.

With the arrival of Rust [121], a memory safe language which requires no expensive memory management, the performance calculus has changed. As a result, many systems which previously used unsafe languages such as C and C++ started exploring Rust [122]–[126]. Redleaf [54] demonstrates the use of Rust to isolate device drivers. Recently, Rust has been added to major OSes including Linux [42], Android [127] and Windows [128]. Most low level kernel software is old and written using C or C++, making the task of porting such software to use memory safe languages, such as Rust, difficult and time-consuming.

2.2.4 Software Capabilities

Software capabilities represent tokens which grant the holder access to objects within the system and can be used to isolate different compartments, whilst still granting bounded and permissions-checked access to certain data. As opposed to hardware capabilities, discussed in Section 2.2.9, software capabilities exist entirely in software implementation, and rely upon software checks to enforce their bounds and permissions. This necessarily results in overhead when compared to implementations which perform the same checks in hardware. Works exploring the use of hardware capabilities to protect within an address space include Mungi [129], Opal [130] and
LXFI [102].

Capsicum [131] extends UNIX APIs by providing primitives which allow processes to be placed in a sandbox mode. From there, they may only access system resources such as the file system using capabilities which have been granted to the process.

2.2.5 Memory Encryption

A less studied but interesting technique is the use of cryptography to secure pointers or regions of memory to prevent unauthorised access and tampering.

Cryptographic extensions to architectures are often utilised for this purpose, to enable more efficient transformation of data. This has been explored by PointerGuard [132], CCFI [133] and MemSentry [134]. Whilst usually implemented in software, memory encryption has also been explored in hardware, for example, Morpheus [135] and MorepheusII [136]. Trusted Execution Environments (TEEs) also make use of memory encryption. TEEs are described in Section 2.2.6.

2.2.6 Trusted Execution Environments (TEE)

Trusted Execution Environments describe technologies which offer confidentially and integrity. In practice, this means that no external, unauthorised access to data is allowed. Additionally, to ensure integrity, unauthorised entities are unable to modify or replace code within the TEE. This creates a shielded execution environment for code and data, allowing applications to be run in scenarios where the host systems software including the OS and hypervisor is untrusted, for example when using cloud providers to run sensitive applications.

TEE was first defined in "Advanced Trusted Environment: OMTP TR1", published in 2009 [137]. Since then there have been numerous commercial implementations of the technology, including ARM TrustZone [71], Intel SGX [138], Keystone [139], MultiZone [140] and Sanctum [141] for RISC-V. These enable *enclaves*. The most widely used implementation is ARM TrustZone, which divides the system up into two worlds, secure and normal. This presents a limitation to fine-grained compartmentalisation, since it results in a maximum of two coarse protection domains. Work has been done to combine TEEs with hardware capabilities [142], which could be used to extend TrustZone to enable more fine-grained isolation. Intel SGX allows applications and OSes to create encrypted *enclaves* to secure components, applications, or even entire OSes.

The use of Intel SGX for compartmentalisation has been studied extensively. Haven [143] and Graphene-SGX [144] run applications within SGX to shield them from the rest of the system. SCONE [145] uses SGX to allow containers to execute securely. SGXBounds [146] provides memory safety for unmodified applications leveraging SGX. Glamdring [147] takes developer annotations for sensitive data, then automatically partitions an application into untrusted and enclave partitions, with the enclave partition then executed in an SGX enclave containing sensitive data and code. SecureKeeper [148] builds upon Apache Zookeeper, using multiple small enclaves to secure user data. Nested Enclaves [149] enable a second tier of isolation within an enclave, through the use of nested enclaves, which are shielded from the rest of the main enclave. CHANCEL [150] sandboxes threads within an enclave to allow an application to securely process requests from different users. Occulum [104] allows multitasking within an enclave, using Software Fault Isolation to isolate between processes.

More recently, another type of TEE has been researched: Confidential Virtual Machines [151]. These enhance the popular Virtual Machine abstraction with the confidentiality and attestability offered by TEEs. Enabled by technologies such as Intel TDX [152], AMD SEV [153], ARM CCA [154] and IBM PEF [155]. Confidential VMs can be used as a mechanism to secure sensitive workloads which are typically run in the cloud. Confidential VMs promise to be more compatible since they are enhanced versions of existing virtual machines and do not require applications to be rewritten, which is necessary to use TEEs. Whereas enclaves focus mainly on process level isolation, confidential VMs focus on entire system isolation. While such technologies offer greater security over traditional virtual machines, they are still vulnerable to attack [156].

2.2.7 Memory Tagging

Memory tagging associates a tag with a portion of memory, with the tag encoded in unused upper bits of pointers. Upon attempted access to memory via a pointer, this tag is compared to that of the memory being accessed. A mismatch will cause the access to fail. This can be used to isolate regions of memory. Modern implementations of memory tagging features include ARM Memory Tagging Extension (MTE) [157], introduced with ARMv8.5, and ADI [158], introduced with the Oracle SPARC M7. The use of memory tagging is increasing with works including Loki [159], Multi-tag [160] and Taxi [161] investigating its use. Furthermore, LLVM provides plugins to accelerate memory tagging using available hardware features [162]. HAKC [163] uses ARM MTE as part of its approach to partitioning the Linux kernel.

2.2.8 Bounds Checking

Bounds checking pointers in hardware is a technique used to constrain memory accesses to within certain bounds allowed for particular pointers. This is often used to prevent common memory safety issues including buffer over- and underflows by preventing such accesses at a potentially low performance cost. These bounds checks can be used to implement isolation between software components.

The most well known implementation is Intel Memory Protection Extensions (MPX). MPX bounds checks pointer accesses by providing instructions to do so. Implementation is optional and relies on a software toolchain to insert checks rather than being performed by default. A number of works have explored the use of MPX, including MemSentry [134]. MPX is no longer included in new CPUs due to numerous performance and security issues identified [164]. Many others have proposed custom hardware, ISA extensions or repurposing of other mechanisms, including AOS [165], Heapcheck [166], Watchdog [167], WatchdogLite [168], Hard-

bound [169] and CHEx86 [170].

2.2.9 Hardware Capabilities

Capabilities can be thought of as a natural evolution of bounds checking mechanisms. They control access to memory not only through bounds, but also setting permissions on what can be performed by a capability which is held. First defined in 1966 by Dennis and Van Horn [171], as "locates by means of a pointer some computing object, and indicates the actions that the computation may perform with respect to that object", capabilities have been explored in systems over the years. Hardware capabilities are examined through the lens of memory protection in this thesis. However, they are in general a higher level concept which can be used to protect many different aspects of a system.

Capabilities were first seen on the Burroughs B5000 [171], [172] in 1961. It implemented a *Program Reference Table (PRT)* containing *descriptors* which located code and data segments in memory and *values* which are scalar data elements. Since these are contiguous regions of memory, the *descriptor* provided bounds. Descriptors also included a bit called a tag, to differentiate between *descriptors* and *values*. All memory references and branches went through the *PRT*.

The Plessey System 250 [173] implemented capabilities as 48-bit values, held in special capability registers, which grant the processor access to objects in the system. A capability consisted of three parts, a base address, a limit and access rights. This allowed bounding and the enforcement of permissions. A capability is always held, which points to a *Central Capability Block*, which defines the execution domain. Critically, the System 250 also had protected procedures, which had their own *Central Capability Block* and were accessed through enter capabilities and arguments were accessed via capabilities. Thus, a protected procedure executed in its own domain protected from the caller, before returning to the caller domain. The Cambridge CAP [174], [175] of the same era also implemented capabilities. Capabilities granted access to objects on the CAP system. Like the Plessey system, it also featured protected procedures.

The M-Machine [176] introduced a concept called *guarded pointers*. These 64-bit pointers, which were used to address a 54-bit virtual address space, encoded a tag to identify a pointer, permissions, and a length in the remaining upper bits. *Guarded pointers* could specify data access, code access, protected entry access or *keys*, which were unforgeable tokens, only accessible by a privileged entity.

The PICA system [177], [178] described an extension of a RISC architecture (MIPS). As well as implementing the well established capability hardware features, protection domain crossings were made simple by the provision of protected procedure calls, which grant access to defined entry points of a protection

domain. Alongside this, instructions to clear registers efficiently made domain switches cheap, with the authors noting only 12-15% performance overhead.

Low-fat pointers [24], building upon work done with fat pointers [25], [26] have been implemented more recently, encoding bounds and a tag to ensure the unforgeability of pointers. Bounds checks are implemented in hardware, with the authors noting a 0% performance penalty and a 3% memory overhead.

More recently, capabilities have been implemented as extensions to architectures including ARM and RISC-V, in the form of Capability Hardware Enhanced RISC Instructions (CHERI [179]) and x86 by CHEx86 [170] and
CODOMS [180]. The use of CHERI and its ARM implementation, Morello, is the focus of this thesis. For this reason, a more in-depth look at CHERI is available in Section 2.6.

2.3 Automation

Automation of the compartmentalisation process is tightly linked with the amount of effort required from the developer. Below are described the main methods for implementing compartmentalisation.

2.3.1 Manual Approaches

Manual approaches rely entirely on developer skill and knowledge to partition an application and implement compartmentalisation policies. While this approach can be the most accurate, it is also the most error-prone [16], with compartments granted too many or too few privileges. In the case of complex applications, some data may be missed, leading to incomplete compartmentalisation. An example of this is modifying an application to use CHERI capabilities in hybrid mode [4].

2.3.2 Guided Manual

In a guided manual approach, the developer is still responsible for code annotations or transformations, however an analysis tool such as a compiler provides hints to a developer by identifying shared data and compartment boundaries, making the task easier [10], [16]. There is less risk of error compared to a fully manual approach.

2.3.3 Policy Based

Whereas manual approaches require direct developer intervention to partition a program correctly, policy based compartmentalisation partitions the application

based on a compartmentalisation policy which has been provided by the developer in the form of annotations. While this still requires the developer to reason about their requirements, the actual work of compartmentalising is done automatically [9], [13].

2.3.4 Automatic

Fully automatic compartmentalisation is challenging to implement well. Inferring precise data dependencies can be difficult to do accurately [181], which means that such tools often need to err on the side of caution and often overshare to avoid breaking applications. However, automated approaches [57], [70] are regarded as the 'holy grail' since they allow compartmentalisation to be applied easily to applications and so increase adoption. Automation is an active field of research.

2.4 Sharing Data

When a legacy application is compartmentalised, data which was previously accessible to the entire address space now becomes inaccessible to other compartments. This is by design but necessitates the use of mechanisms to restore limited data sharing between compartments. This is important to pass function arguments and return values amongst other data. Data sharing often involves some form of message passing between compartments. For example, for applications partitioned into separate processes, this could take the form of inter-process communication (IPC) [182].

In addition, some mechanisms provide specific communication methods. For example, CHERI capabilities [179] can be used in place of pointers to pass arguments between compartments.

2.5 Compartmentalisation In Practice

Having described the theoretical principles and practical mechanisms of compartmentalisation, the following sections now move on to consider real implementations of compartmentalisation and answer the question: *Why is compartmentalisation not mainstream?*

2.5.1 Deployed Software

Firefox uses RLBox to isolate itself from its libraries [10]. Libraries are placed into sandboxes. This has been used in deployed Firefox to sandbox performance sensitive web page decompression libraries, including libGraphite. RLBox is a

system which aids in the isolation of applications from their libraries by providing a simple API. It ensures that data-flows between compartments are sanitised by flagging such data-flows during compilation, creating a feedback loop for developers to fix these compile time errors. Isolation is implemented either through process based isolation, or SFI. In production Firefox, a WebAssembly based SFI sandbox is implemented. Analysis showed that by sandboxing libGraphite in production, an 85% performance penalty was applied that that code, which translated to a 50% slowdown for the renderer overall [10].

OpenSSH has implemented compartmentalisation techniques for over two decades. The first privilege separation was introduced in 2002, to separate the privileged server process from an unprivileged user authentication process [3]. The privileged server acts as a monitor of the unprivileged user process. OpenSSH also implements sandboxing of potentially unsafe code such as pre-authentication checks which could be exploited by an attacker. The sandboxes can use a variety of mechanisms including SELinux [183] and Capsicum [131] to achieve this.

By implementing privilege separation, two thirds of the source code executes without privileges, including third party libraries such as *zlib*. Due to the lack of data copying, privilege separation is found to not impose a performance penalty [3].

2.5.2 What Is The Problem?

Despite having described many different approaches to compartmentalisation and many different trust models, compartmentalisation is still mostly a toy of academia and is not widely used in applications. There are a number of reasons for this. Firstly, many mechanisms impose a performance overhead, which is simply too high for many [8], [184]. Next, as well as a potential performance penalty, the cost of retrofitting applications which were not designed with compartmentalisation in mind, can be prohibitive without effective automation [13], [70]. Finally, certain mechanisms impose limitations [93], [185], such as a limited number of compartments available with MPK. Such limitations have stopped compartmentalisation from effectively being integrated into everyday software and can be summarised as: **too slow**, **too hard** and **too limited**.

2.6 CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) is a set of extensions for existing RISC architectures including MIPS, RISC-V [179] and later ARMv8 (the implementation is called Morello) [21]. CHERI adds hardware capabilities, described in Section 2.2.9, to existing architectures. Morello has since been implemented in hardware [186] based on an ARM Neoverse N1 SoC [187]. This section,

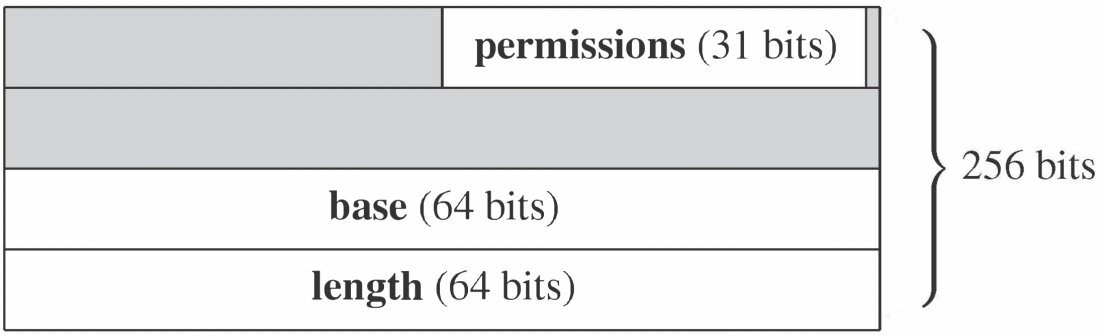

Figure 2.4. The original encoding of a CHERI capability. This image is taken from *The CHERI capability model: Revisiting RISC in an age of risk* [22].

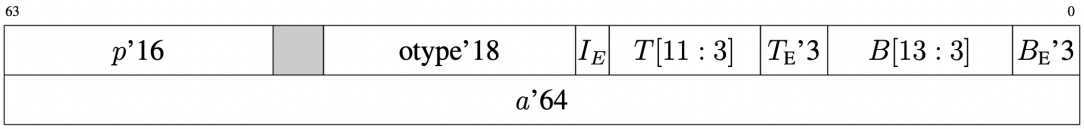

Figure 2.5. The CHERI concentrate capability encoding. This image is taken from *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)* [179]. a: pointer address, B & T: base and top related to bounds, I_E & B_E & T_E: exponent bits, *otype*: object type, p: permissions.

first introduces CHERI concepts before moving on to a detailed study of the state-of-the-art compartmentalisation works using CHERI, in order to understand how the contribution described later in this thesis fits in to the existing body of work.

2.6.1 An Introduction To CHERI

CHERI was first introduced by Woodruff *et al.* [22] in 2014. Originally implemented using MIPS [188], it has since been expanded to cover RISC-V and ARMv8 [189]. Since its first introduction, the ISA has also been refined. For example, capabilities were originally encoded using 256-bits on 64-bit architectures, displayed in Figure 2.4. With the introduction of the CHERI concentrate encoding [190] which is shown in Figure 2.5, capabilities can now be encoded in 128-bits for a 64-bit architecture, by utilising floating point encoding, however, not all memory bounds can be represented by this meaning that memory padding could be required.

Capabilities

At a high level, a capability can be thought of as a pointer with additional bounds and permissions information attached to it. The result is that an attempted memory access is automatically and atomically checked against these bounds and permissions in hardware to establish whether it is legal. This can be used to enable compartmentalisation in a single address space by restricting the bounds and permissions available to compartment capabilities to a limited portion of the address space.

CHERI capabilities are encoded using 128-bits on 64-bit systems and 64-bits on 32-bit systems. A capability encoding contains a 32- or 64-bit integer *base value*, which is equivalent to an integer pointer. This is augmented with bounds information in the form of a limit or a length. Bounds information specifies starting at the base what the allowable range of addresses is. Additionally, each capability stores an *object type*, *permissions* and a *tag*.

Since capabilities are double the size of regular pointers, special capability registers must be used to store them. This can either be implemented as a separate register file (CHERI-MIPS) or can be an extension of current general purpose registers providing an additional capability view. The latter is the implementation present on the Morello platform.

In addition to general purpose capability registers, several other registers are extended to enable capability support. These include the *Capability Stack Pointer (CSP)* which can be used to restrict the stack pointer via a capability, and the *Program Counter Capability (PCC)* which extends the program counter to bound instruction fetches, acting as a restriction on the code which can be executed. A *Default Data Capability (DDC)* register is also provided to restrict non-capability memory accesses.

Guarded Manipulation & Monotonicity

CHERI enforces *monotonicity* upon capabilities, which means that new capabilities may only be derived from existing capabilities via special *guarded manipulations* which prevent a capability from increasing its bounds, it may only narrow them, and prevents capabilities from obtaining additional permissions, it may only reduce permissions. These properties prevent capabilities being manipulated and undermining the protection they offer.

Capability Tag

Each capability has a tag associated with it. The tag indicates whether a capability is valid and can be dereferenced (tag set). If the tag is cleared, which can result from attempting to modify or use a capability illegally, then the capability cannot be dereferenced. Capability registers are 129-bits wide to accommodate the tag in bit 128. However, a capability stored in memory is stored separately from its tag to prevent forgery. The tag is architecturally invisible to the programmer. Any attempt to manipulate a capability in memory directly as opposed to using capability instructions will clear the tag associated with that location. When loaded into a register, the associated tag bit is automatically loaded alongside.

Capability Permissions

Capability permissions indicate what a capability can be used to do. This also includes permissions on the *PCC* and *DDC* capabilities. The capability permissions implemented on Morello are listed in Table 2.1. Restricting capability permissions can be used to enforce lower privileges for compartments, for example, removing the ability to access system registers.

Table 2.1. Permissions available in Morello.

Permission	Purpose
Global	Allow capability to be stored via capabilities that do not have StoreLocalCap permission set.
Executive	Code is executed in executive mode, if this permission is not set, execution takes place in restricted mode. These modes have different views of the same global registers (register banking).
Load	Allows load via capability.
Store	Allows store via capability.
Execute	Allows code execution via capability.
LoadCap	Allows a capability to be loaded via this capability.
StoreCap	Allows a capability to be stored via this capability.
StoreLocalCap	Allows a capability to store a capability not marked as Global.
Seal	Allows a capability to be used to seal another capability with an object type set to be equal to the value of this capability.
Unseal	Allows this capability to unseal a capability with an object type equal to this capability value.
System	Allows access to system registers.
BranchSealedPair	Can be used by a branch sealed pair instruction.
CompartmentID	Can be used as a compartment ID.
MutableLoad	A capability loaded via a capability without MutableLoad set will have its Store, StoreCap, StoreLocalCap and MutableLoad permissions cleared.
User	Software defined.

Capability Sealing

A sealed capability is immutable and non-dereferenceable meaning that any attempt to use a sealed capability will result in an exception. Sealed capabilities can be used in compartmentalisation to facilitate compartment switches, which can be done via sealed entry capabilities, for example granting access to a privileged compartment switcher, without being able to dereference such a sensitive capability arbitrarily. A sealed capability can only be modified after it has been unsealed via a permitted instruction or capability matching its object type.

Capability Object Type

An object type is metadata which is set when a capability is sealed. It identifies sealed capabilities which can be used together. When unsealing a capability, its object type is compared to the value of the unsealing capability and unsealing is

only permitted if they match. In this way, the object type is used as a key to ensure a capability is unsealed using only the correct capability.

Hybrid Mode Execution

CHERI can be used in two different modes of execution: hybrid mode and pure capability mode. In pure capability mode, all pointers are replaced by the compiler with capabilities, hence the name pure capability mode. In hybrid mode, capabilities and capability instructions can be used alongside regular integer address instructions and integer pointer addresses. This is done to ease the porting of existing code. For example, a regular operating system can be run and incrementally components can be ported, whilst running alongside un-ported components. This promises to ease the integration of compartmentalisation using CHERI capabilities into existing code bases.

In hybrid mode, all regular data load and store instructions are bounds checked against the *DDC*, meaning that to enforce compartments in hybrid mode, the *DDC* and *PCC* capabilities are dynamically restricted to cover the code and data needed for a compartment.

2.6.2 CHERI State Of The Art For Compartmentalisation

The following sections examine the current state-of-the-art compartmentalisation work using CHERI.

CheriRTOS

CheriRTOS [191] is a port of the real time operating system FreeRTOS for embedded devices. It utilises CHERI-64: 64-bit capabilities, which are suitable for 32-bit devices such as embedded devices. CHERI-64 is propsed as a way to address the limitations of typical MPU based approaches. Compartmentalisation is in hybrid mode, where the *DDC* and *PCC* are used to enforce bounds on the code and data a task can access. The hybrid mode approach is used to enable greater compatibility, with the paper claiming that only minor modifications were necessary to run applications. CheriRTOS implements compartmentalisation at task granularity which is similar to a process on a UNIX-like operating system, making isolation quite coarse.

Compartments can communicate either via message passing, which has a high performance cost, or compartment switches, which can be initiated by a compartment. Compartment switches are achieved by making a set of sealed capability pairs available to a compartment to unseal and install, thus switching compart-

ments. Additionally, the heap is secured by a separate compartment which handles memory allocations such as malloc and returns bounded capabilities.

CompartOS

CompartOS [6] is a compartmentalisation framework which is designed for embedded devices, however the authors make the distinction that it is targeted at high-end embedded devices. CompartOS removes the need for any MMU or MPU based protection relying entirely on capabilities, much like CheriRTOS. It instead enforces isolation between linkage modules automatically and compartmentalisation extends to all aspects of the system, including access to system resources. In addition, a major emphasis is placed on availability and recovery from failure. To that end, each compartment has its own software fault handler.

Compartmentalisation granularity is specified at development time, for example, this could be between individual source code files, compiled into relocatable libraries, which are loaded and isolation is enforced between these linkage modules. Only the secure loader is required to enforce compartmentalisation, disposing of the need for a UNIX-like process model, and notions such as kernels and syscalls. All accesses are performed via capabilities. Compartmentalisation is in pure capability mode.

While the automated linkage based compartmentalisation eases the adoption and porting effort required, it does not take full advantage of the byte granularity isolation which is possible with CHERI.

CherIoT

CherIoT [31] is an iteration of the CHERI-64 specification which is optimised for embedded devices with the purpose of simplifying the implementation and so saving cost and die area in cost sensitive IoT devices. Implemented using 32-bit RISC-V, it omits several CHERI features including hybrid mode, opting instead to support only pure capability mode. This simplification allows for the ommission of the *DDC* since this is not needed for pure capability mode. Compartmentalisation is done in pure capability mode.

CheriOS

CheriOS [30] is a pure capability microkernel which follows the traditional microkernel goal of presenting a small attack surface whilst also granting the microkernel itself lower privileges. It does this by implementing a small (<3000 assembly instructions) 'nanokernel' which is responsible for mediating access to the architecture, memory management and other critical functionality. The report suggests

that this could be implemented by a CPU vendor as firmware, although it has been implemented as a hypervisor so far.

The CheriOS microkernel is subsequently left to handle scheduling, interrupts and message passing between applications. The result is a much reduced trusted computing base (TCB), since the microkernel is no longer assumed to be trusted. The use of capabilities enforces strong memory safety and isolation, and CheriOS operates on a model of mutual distrust, which includes the microkernel but not the nanokernel. Compartments in CheriOS are at the process level.

CheriOS implements concepts to allow distrust between compartments: *reservations* which allow distrust of delegated memory by ensuring that the delegator cannot access it at the same time, and *foundations* which enable attestation.

However, CheriOS lacks POSIX compliance, making its usefulness limited. While abstraction layers do exist to provide compatibility, these hurt the performance of the system and so are a compromise.

CheriBSD

CheriBSD is a widely used port of FreeBSD which has support for both CHERI RISC-V and ARM Morello in hybrid and pure capability modes. CheriBSD is a good showcase for the utility of incremental porting. Watson *et al.* described a hybrid capability approach to CheriBSD in 2015 [4]. A hybrid mode OS design was described in which the kernel runs in hybrid mode, facilitating the execution of capability unaware, hybrid mode and pure capability applications. In their implementation, applications compartmentalised using the lib_cheri api, were explored. lib_cheri enables compartmentalisation in pure capability mode within a single address space. The kernel was only minimally extended to support capability register context switching and tagged memory. Since then, the CheriBSD kernel has been fully ported to use capabilities but still runs in hybrid mode, detailed by Davis *et al.* [192] in 2019. However, CheriABI pure capability user-space applications are now supported.

CAP-VMs

CAP-VMs [7] (cVM) introduces a virtual machine-like abstraction which does not rely on virtualisation and addresses common limitations of container and VM based approaches to isolation. Hybrid mode global *DDC* and *PCC* capability pairs are used to isolate application components within a single address space. Running a LibOS within a cVM, it is possible to run multiple application components alongside each other. Communication between compartments is enabled though asynchronous read/write buffers, streams and direct calls to switch between compartments.

Hybrid mode is used to improve compatibility, with the only changes needed to support the proposed system being changes to use the data sharing API. Application components communicate remotely via communications APIs already, making it simple to share data between them in hybrid mode. All isolation and management of cVMs is handled by an *intravisor*, which is responsible for functions such as creating and installing capabilities for cVMs. In addition to the sandboxing enabled by the restricted *DDC* and *PCC* bounds, the permissions granted to cVM capabilities are also heavily restricted. Isolation is provided between an application component and the LibOS, as well as between the cVM and the intravisor.

CHERI JNI

CHERI capabilities have also been explored in the context of java native code [29]. Unlike java byte code which is executed by the java virtual machine and is considered memory safe, java native code is included often as C libraries which are executed directly and offer no memory safety. Placing these portions of code into sandboxes isolates a major source of vulnerabilities. Sandboxes are implemented as pure capability code to ease the implementation of the java native interface (JNI), however the authors note that where native code cannot be modified, a hybrid approach may be considered to improve compatibility. Buffers are accessible to JNI code via capabilities, which often have only read-only permissions.

Research Opportunities

By examining the current body of research utilising CHERI, a number of patterns become clear. Firstly, hybrid mode is used where compatibility is desired due to the ability to run legacy, capability unaware instructions alongside capability instructions. Hybrid mode compartmentalisation within a single address space is under-explored in the literature, considering the promises of low engineering effort and easy compartmentalisation.

Additionally, the arrival of Morello as a hardware implementation opens up the opportunity to gain performance insights by evaluating on real hardware as opposed to the FPGAs and softcores primarily used in the literature. Table 2.2 compares existing work to the investigation proposed in this thesis.

2.7 Unikernels

Unikernels [15], [193]–[204] are a recent model of Operating System (OS) in which a single application is statically compiled along with its library dependencies as

Table 2.2. Comparison of compartmentalisation using CHERI. *SAS* = Single Address Space Compartmetalisation, *EOH* = Evaluated On Hardware

Work	Mode	SAS?	ISA	EOH?
CheriRTOS	Hybrid	No	CHERI MIPS	No
CompartOS	Purecap	Yes	CHERI RISC-V	No
CherIoT	Purecap	Yes	CHERI RISC-V	Yes
CheriOS	Purecap	No	CHERI MIPS & CHERI RISC-V	No
CheriBSD	Purecap	Yes	CHERI MIPS, CHERI RISC-V & ARM Morello	No
CAP-VMs	Hybrid	No	CHERI RISC-V	No
CHERI JNI	Purecap	No	CHERI MIPS	No
This Thesis	**Hybrid**	**Yes**	**ARM Morello**	**Yes**

well as a very thin OS layer. Unikernels are typically executed as a small virtual machine on top of a hypervisor. The resulting executable is a highly specialised OS, which is both fast and lightweight (low memory and disk footprint and fast boot times). Additionally, as a result of their specialised nature and their removal of unnecessary code, unikernels present a reduced attack surface when compared to traditional monolithic operating systems. To improve performance, all of the kernel and application code resides in a single address space with no isolation between the application and the kernel. While beneficial for performance reasons, this can be a security issue which can be mitigated with compartmentalisation [32].

Using a unikernel allows for the exploration of compartmentalisation with CHERI in hybrid mode, within a single address space. Additionally, it will also allow lessons to be drawn which can be applied to other single address space applications of CHERI in hybrid mode.

2.7.1 FlexOS

FlexOS [15], [205] is a unikernel which is based upon Unikraft [193]. It is designed with flexible security policies in mind. A developer annotates function calls which cross compartment boundaries and annotates data which should be shared. At build time, the FlexOS toolchain then transforms these annotations into mechanism-specific code based on a supplied configuration file. This means that the precise isolation mechanism can easily be changed in the future. Additionally, compartment boundaries are easily set and moved simply by supplying a different compartment configuration file to the toolchain. This makes it an ideal candidate to implement a CHERI backend on Morello because 1) it is a lightweight unikernel with a small code base and 2) it is not tied to a specific isolation mechanism, allowing it to be extended. Isolation backends have been implemented using Intel MPK and Intel EPT.

2.8 Summary

This chapter has motivated research into compartmentalisation as an important technique, which is needed to isolate sensitive components of computer systems, thus decreasing the damage which can be wrought by an attack or malfunction, accidental or malicious in nature. Next, the mechanisms which can be used to implement compartmentalisation were examined and real world examples of compartmentalised software were described, observing that despite the plethora of research done, compartmentalisation remains firmly in the realm of academia rather than commonplace considerations for commercial software deployments. Following this, the state-of-the-art research using CHERI was presented, finding that little work has explored the use of hybrid mode in the context of single address space application compartmentalisation, despite the potential advantages offered by such an approach.

The remainder of this thesis explores the use of CHERI capabilities in hybrid mode, explored in Section 2.6, as a way to trade off the performance and engineering costs and security needs of compartmentalisation, by being compatible with existing software and enforcing bounds and permissions checking in hardware. FlexOS, a security oriented unikernel is extended with a hybrid mode CHERI backend to explore different models and approaches to compartmentalisation, including data sharing.

Chapter 3

Design

The following chapter first examines the requirements which must be addressed with the design for CHERI compartmentalisation in hybrid mode on Morello: low performance cost, low engineering cost and good scalability. Following this, a high-level overview of the design of such a system is given. This is followed by the proposal of five different approaches to data sharing. Based on the requirements, defined in Section 3.1, the most suitable data sharing mechanisms will be selected for evaluation. This chapter focuses on **RQ1**: addressing which compartment models are possible using Morello in hybrid mode.

3.1 Requirements

The following are requirements which must be carefully evaluated throughout the rest of this thesis, in relation to the design decisions and implementation of compartmentalisation using CHERI hardware capabilities in hybrid mode. They affect how easily the compartmentalisation methods proposed in this thesis can be applied to real software.

3.1.1 Low Performance Cost

Implementing compartmentalisation will introduce overheads due to the need to share data in a controlled manner, and also the need to perform compartment switches where previously standard function calls were. However, different configurations, use-cases and workloads will accept different overheads. In evaluating potential solutions, it is important to consider the associated performance cost. The design of the system must also seek to minimise the performance cost of components such as the compartment switching mechanism while achieving a high level of security.

The performance cost will be evaluated against the DARPA CPM requirements for compartmentalisation [27], which state that an overhead of $< 15\%$ for OS level isolation and $< 5\%$ for application level isolation is acceptable.

3.1.2 Low Engineering Cost

Solutions must make the cost of retrofitting compartmentalisation into legacy software as low as possible, and this cost must be weighed up against the performance requirements and security needed for the system. For example, if high performance is critical, it may be possible to implement compartmentalisation which requires higher engineering effort but affects performance less and still gives the required security guarantees. The cost of engineering extends not only to the new programming abstractions or annotations which are implemented and must be integrated into a complex code base, but also includes the introduction of a trust model which must be integrated into legacy software.

Returning to DARPA CPM requirements for compartmentalisation [27], the maximum engineering effort is deemed to be $< 2\%$ of the code base annotated for OS level compartmentalisation. For the application level, $< 0.2\%$ of the code base annotated is deemed acceptable.

3.1.3 Scalability

Scalability concerns stem from both performance and engineering costs and can affect how many compartments can be implemented using a particular design, for example the performance or engineering costs increase with each additional compartment. This is particularly important in a unikernel where high performance is helped by having no isolation between the application and kernel. For this reason, selecting designs which scale well for their use-cases is important.

3.2 Design Overview

FlexOS [15] has been chosen to implement a prototype system for evaluation. FlexOS' design enables the easy compartmentalisation of legacy code, having been designed with isolation and security as primary concerns. Additionally, FlexOS is implemented in a way which uncouples the compartmentalisation abstractions, such as call gates, from the implementation, reducing the burden of implementing a new CHERI backend.

Unikernels [119], described in Section 2.7, run a single application in the same address space as the core kernel components, resulting in no isolation between the application and kernel. Compartmentalisation in this context is intra-address space, meaning that new trust boundaries must be established within existing software and between previously mutually trusting components such as functions or core OS libraries, as well as within the application itself. A unikernel is used for the prototype because the small code base eases the engineering burden of integrating compartmentalisation into an OS.

FlexOS is based upon a POSIX compatible unikernel, Unikraft [193], which will greatly improve the adoptability of the system, with widely used applications such as SQLite, available and compatible. This also enables easier comparison to other isolation mechanisms; FlexOS has previously been evaluated with both MPK and EPT mechanisms. Since unikernels such as FlexOS and Unikraft are highly specialised, they also improve performance by minimising the inclusion of unnecessary code and libraries. This also serves to reduce the attack surface presented.

Compartments are defined statically at build time in the linker script, using a configuration file provided to the build tool by the developer [15]. The boot code initialises global compartment capabilities based on these defined boundaries. Once the developer has defined a compartment scheme, gates are inserted in place of function calls, where these calls now cross compartment boundaries. Gates are the beginning of the switching process, and are used to invoke the switcher, which then performs the compartment switching, including switching the stack and global compartment capabilities. Finally, a trampoline function is called, which branches to the desired function. The switcher and global compartment capabilities cannot be accessed by compartments directly, instead they must use a sealed capability. The switching mechanism is kept as lightweight as possible to minimise overhead while preserving security. The compartment switching mechanism is described in more detail in Section 4.2. All data is treated as private compartment data by the build tool, unless it is specifically annotated and thus shared. Below, the possible solutions for sharing data are examined.

The trusted computing base (TCB) includes the switcher, gates, trampoline, early boot code including capability initialisation code, memory manager, scheduler and interrupt handler.

3.3 Approaches To Data Sharing

Most data are private to a compartment because it resides in its region of memory, which other compartments cannot access. Only explicitly shared data, are accessible to other compartments. Pointers to data, which are now private to a compartment, cannot be dereferenced successfully by design. However, this raises issues for most compartment models: *how to selectively share data between compartments?* Pointers to data are used in many applications and libraries as a way of avoiding the cost of copying data when arguments are passed, which is done for performance reasons. The different approaches to restoring data sharing between compartments are examined, and their merits are evaluated in terms of performance and engineering costs, as well as their scalability.

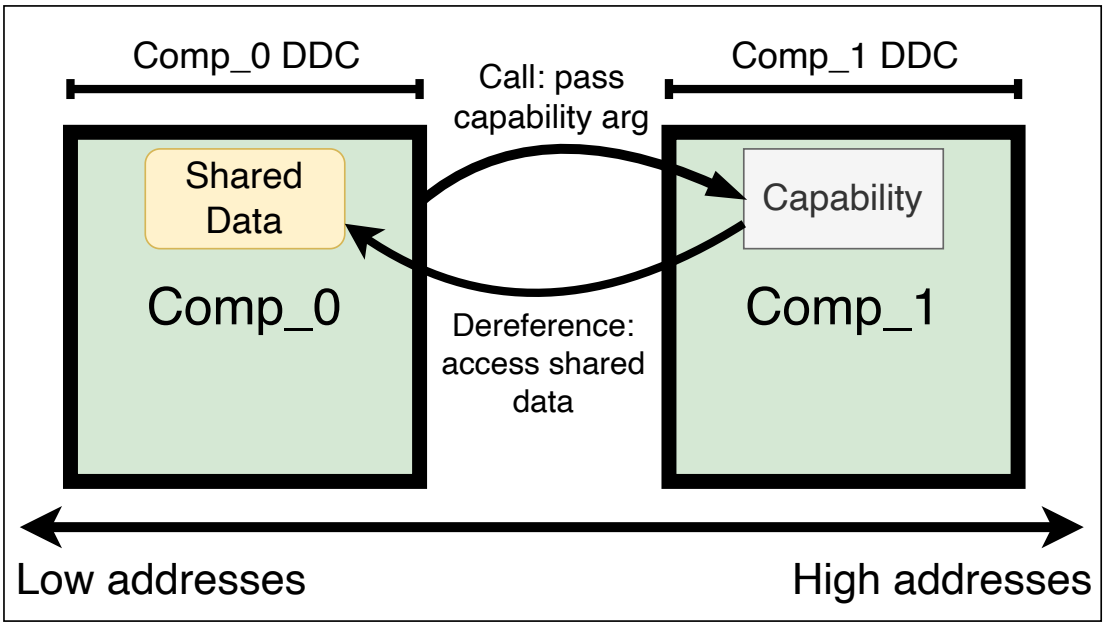

Figure 3.1. Example of data accessed via a capability. Here, the shared data is in compartment 0, which is unreachable from compartment 1. A capability is passed as an argument when compartment 1 is called. The data is then accessed by dereferencing the capability.

```
struct stat *statbuf; // pointer
struct stat *__capability statbuf; // capability
```

Listing 1. Example of compiler annotation needed to change pointer to capability.

3.3.1 Manual Capability Propagation

Capabilities grant controlled, bounded access to data which is not within bounds of a compartment *DDC*. This removes the need to share data through other mechanisms, such as shared data regions, and eliminates the potential for oversharing data when compared to coarse grained shared memory where all shared data may be accessed. Pointers can be changed to capabilities by using compiler annotations (Listing 1). Figure 3.1 illustrates this in practice. Here, the shared data is in compartment 0, which is unreachable from compartment 1. A capability is passed as an argument when compartment 1 is called. The data is then accessed by dereferencing the capability, with compartment 0 effectively lending the shared data to compartment 1. This approach could be extended to an unlimited number of compartments, since capabilities are used in the same way as pointers. In terms of security, the compartment is isolated by the *PCC/DDC*, constraining all non-capability operations made by the compartment, and shared data is tightly bounded by argument capabilities, resulting in strong isolation.

Trust Model

Here, a compartment represents the code and data of an untrusted function, and is isolated from the rest of the system by way of a sandbox. Pointer arguments entering the compartment are replaced with capabilities. The rest of the system

```
1  void foo(mystruct *stat, int index) { // Original
2    char *str = stat->str;
3    bar(str);
4    float *element = stat->array[index];
5    float *next = element+1;
6  }
7
8  void foo(mystruct *__capability stat, int index) { // Ported
9    char *__capability str = stat->str;
10   bar((__cheri_fromcap char*)str);
11   float *__capability element = stat->array[index];
12   float *__capability next = stat->array[index+1];
13 }
```

Listing 2. Example of a simple function annotated to use capabilities.

is trusted and can access any memory within the sandboxed compartment.

Engineering Cost

A developer must identify pointers which cross compartment boundaries and apply annotations to transform the pointers to capabilities. At this point, functions called, which previously accepted pointer arguments, must now be modified to accept capability arguments as shown in Listing 2. Following this, uses of the new capability argument such as variable assignments much be updated. Additionally, some code may need to be rewritten to take into consideration capability monotonicity. This is illustrated in Listing 2.

Capabilities flowing out of the compartment must be changed back into pointers with a cast (line 10). Capability monotonicity must also be respected. For example, a capability cannot extend the bounds of the capability it is derived from: `element+1` (line 5) is forbidden because it refers to memory outside the bounds of `next`, and that code must be redesigned. Other types of changes may be needed depending on the ported code [206].

This solution is only amenable to small scenarios such as compartmentalisation at the function level because the development cost of mixing pointer and capabilities can become burdensome. Consider the following scenario which was encountered when compartmentalising SQLite: `Compartment 1: SQLite, Compartment 2: vfscore + ramfs, Default Compartment: everything else`. Now consider the path of execution shown in Figure 3.2. In this scenario, a struct is allocated on the stack in compartment 1, a pointer is then passed through compartment 2, and used in the default compartment (`memset`). This would require the developer to modify a libc function to now accept a capability argument. To do so would also require rewriting all other calls to `memset`. The example given demonstrates that this problem is not simply one encountered when attempting to compartmentalise libc. Libc is in the default compartment and yet data flows between compartments can mean that capabilities flow to many functions, which explodes the engineering effort which is required.

```
findInodeInfo (SQLite, Comp1)
        ↓
fstat (vfscore, Comp2)
        ↓
sys_fstat (vfscore, Comp2)
        ↓
vfs_fstat (vfscore, Comp2)
        ↓
vn_stat (vfscore, Comp2)
        ↓
memset (libc, Default)
```

Figure 3.2. Path of execution, demonstrating the difficulty faced when manually annotating pointers to capabilities. `Compartment 1: SQLite, Compartment 2: vfscore + ramfs, Default Compartment: everything else`.

Scalability

Although the increasing use of capabilities alongside regular integer pointers can increase complexity, this cost does not necessarily increase with the number of compartments, rather the complexity grows regardless of the number of compartments. For this reason, if small or simple compartments are designed, the engineering burden is lower.

Performance Cost

Replacing pointers with capabilities has minimal overhead when compared to other methods. No additional data copies are required, such as copying data to a shared data region of memory or transforming stack data allocations into dynamic allocations. It is also cheaper than relying on additional call wrappers to enable capability access to data as is described in Section 3.3.5, since no additional instructions are required, only capability aware replacements. The use of capabilities in place of 64-bit pointers can also result in increased cache pressures since the cache size is unchanged when adding support for capabilities in hardware, but pointers now occupy twice the cache real-estate.

3.3.2 Overlapping Shared Data Region

This approach is similar to the one used with FlexOS MPK [15]. Here, a region of memory is reserved by the developer at build time to contain shared data. For static variables are annotated by the developer to relocate them at link time. The

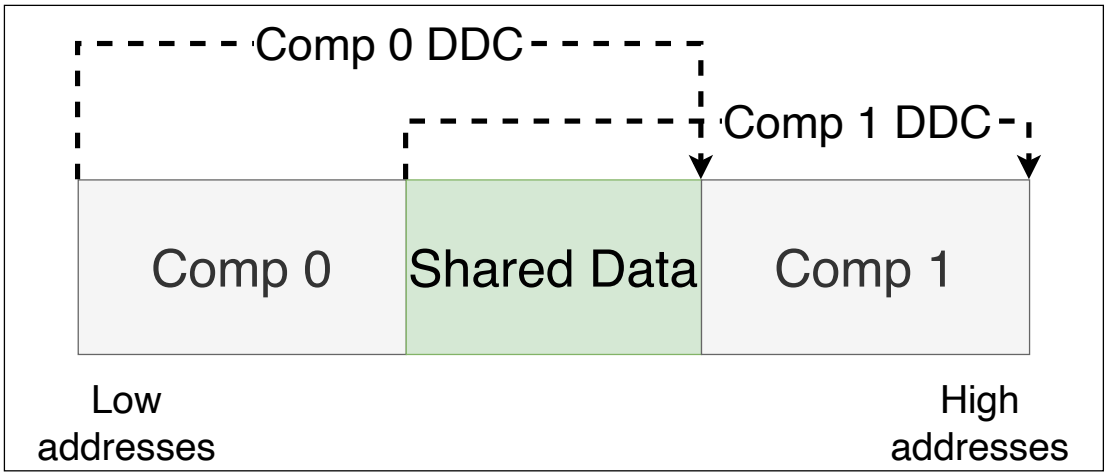

Figure 3.3. Compartment bounds when a shared memory region is used. The compartment bounds overlap to encompass shared data. The grey boxes represent compartment private data.

```
void foo() { // Original
  int x;
  bar(&x);
}

void foo() { // Ported
  int __flexos_shared x;
  __flexos_gate(bar, &x, compartment1);
}
```

Listing 3. Examples of FlexOS annotations.

developer also annotates dynamic memory allocations to use a shared memory allocator, which allocates the data into shared memory. Since capability bounds must cover contiguous memory, the shared data region is located between pairs of communicating compartments in memory. Their *DDC* bounds now cover this region, granting both access to shared data, illustrated in Figure 3.3. Data sharing is at a much coarser level than when using manual capability propagation, since those accesses are tightly bounded. In contrast, all data in shared memory is accessible to a compartment, even if not all data in shared memory is relevant to a particular compartment.

Trust Model

Mutual distrust is enforced between compartments, with none able to access the others' private data. Data must be specifically shared. Shared data is accessible in shared memory. All compartments can access all data within the shared memory which they have access to.

Engineering Cost

With this approach, shared data needs to be marked as such with annotations in the source code. The function calls at compartment boundaries must similarly be annotated. This is illustrated on Listing 3, where `foo` and `bar` are placed

in different compartments. Code transformations performed by the build tool, use these annotations to automatically allocate shared data in memory which is accessible from all compartments, and to instantiate gates. The engineering cost of this approach is relatively low because only annotations are required, rather than source code rewriting. Compared to the manual capability propagation, the effort is much lower, because the annotations needed to share data are needed only at declaration, whereas all pointers on the path of shared data using capabilities must be annotated and source code rewritten.

Scalability

This approach may only work for two compartments due to the contiguous bounds requirements of capabilities. The specific use case is important in determining if this approach is limited to two compartments only. As an example to demonstrate this, a developer may choose to implement three compartments, however in this model, compartments 1 and 2, and compartments 2 and 3 share data. Such a scenario could apply an overlapping shared data region to more compartments, however, it is not a generic solution and would require extensive data flow analysis to determine its applicability. A scenario where a compartment needs to share data with multiple other compartments could not work because compartment bounds will need to encompass other compartments in order to also overlap with shared memory.

Performance Cost

A performance cost associated with this approach relates to shared stack variables, which must now be allocated on a heap in the shared memory, with the associated cost of an allocation.

3.3.3 Shared Data Capability

Access to shared data is limited by the contiguity requirements of compartment bounds, making the approach only applicable to compartment pairs which share data only between them, where the exact data flows allow bounds to overlap safely. However, rather than overlapping a compartment *DDC* with shared data, a separate capability granting access only to the shared data could be loaded into the *DDC* register or used with a capability aware instruction to temporarily grant access. Two methods are proposed for using a shared data capability. These are exception based shared data access and load/store macro based shared data access, where C macros are used to perform capability loads and stores from shared memory. The two methods trade off performance and engineering effort.

To avoid additional memory accesses, the shared data capability is kept in a compiler reserved register. Since these are shared data methods, the data sharing is coarser than when using manual capability propagation.

3.3.4 Exception-Based Shared Data Access

Upon attempted access to shared data via a pointer, a capability bound fault is triggered, at which point the compartment *DDC* and shared data capability are swapped, granting access to shared data. When compartment data is accessed, the same happens. This is illustrated in Figure 3.4.

Trust Model

Mutual distrust is enforced between compartments, as with other shared data approaches. Data must be shared, otherwise it is private. When exception based access to shared data is available, all shared memory is accessible to compartments.

Engineering Cost

The engineering cost associated with this approach is lower than that associated with the overlapping shared data region approach, since the same annotations are required, but it is not necessary to reason about which compartments pairs share data. Therefore, it is an approach which requires low effort.

Scalability

Unlike the overlapping shared data region approach, this approach will scale to an unlimited number of compartments, requiring the same effort for each additional compartment. This is because the *DDC* is switched to one which covers the shared data region, and so it does not matter where this resides.

Performance Cost

Relying on exceptions lowers the porting cost by enabling the use of the same shared data annotations as are used in the overlapping *DDC* approach. However, relying on exceptions is a costly approach [207] and similar approaches such as the trap and map approach taken by CubicleOS [14] proved expensive.

3.3.5 Load/Store Macro-Based Shared Data Access

In contrast to the automated runtime switching enabled by exception handling, a developer can manually wrap shared data accesses in special macros which perform

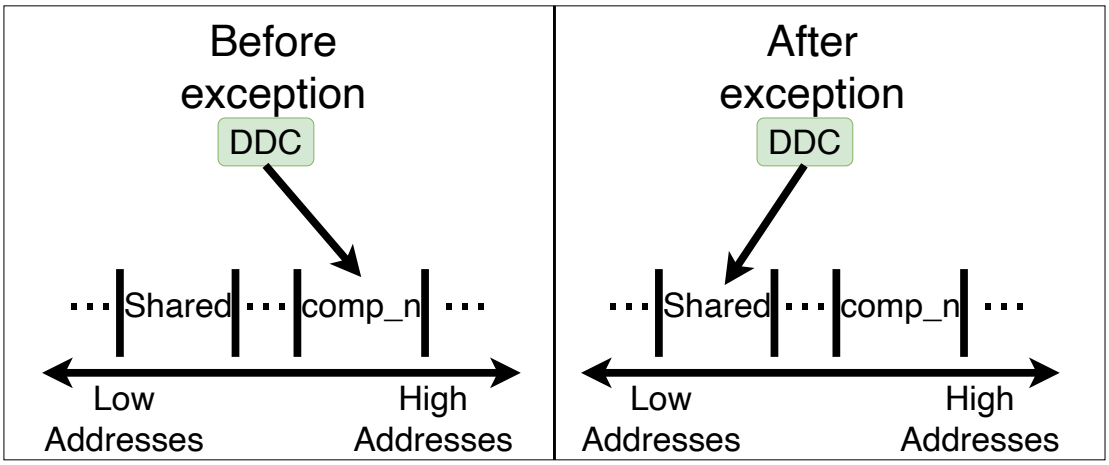

Figure 3.4. Exception based shared data access approach. *DDC* before and after a capability bounds fault when attempting to access shared data.

loads and stores via the reserved register containing the shared data capability. In this way, capability loads and stores can be performed in the shared data memory without needing to overlap the *DDC* bounds.

Trust Model

Mutual distrust is enforced between compartments, as with other shared data approaches. Shared data is accessed via capability loads and stores, meaning that all shared data is accessible to all compartments.

Engineering Cost

The engineering effort required is higher than other shared data approaches as a result of wrapping all shared data accesses. Where frequent shared data accesses occur, the engineering cost is higher. This burden could be reduced by implementing automated data flow analysis to flag such accesses to developers. Indeed, the process could be entirely automated by instrumenting such accesses with the relevant capability aware load and store instructions.

Scalability

The cost of wrapping shared data accesses does not increase with additional compartments, meaning that it can scale to as many compartments as are required.

Performance Cost

This approach does not suffer the performance cost of repeated exception handling but may require additional instructions, meaning that the performance cost will be higher than the overlapping shared data region approach.

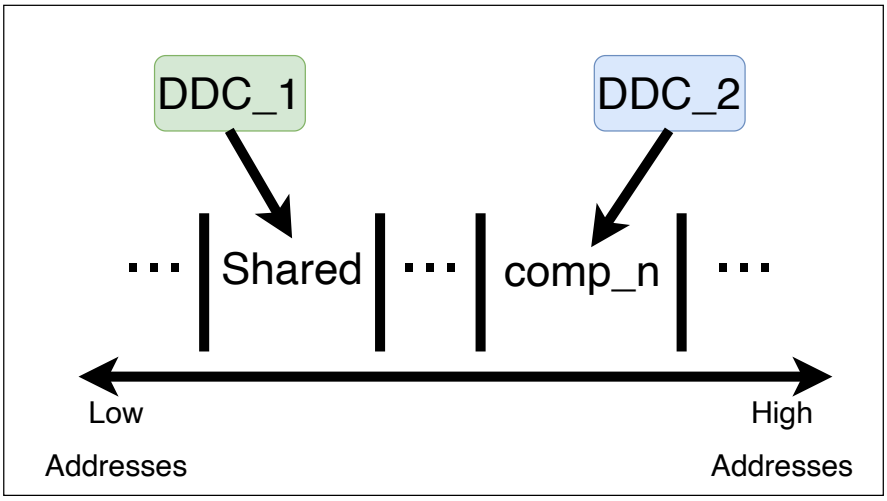

Figure 3.5. Multiple *DDCs* which are initialised to cover different memory regions, thus avoiding the need to switch.

3.3.6 Shared Data With Multiple *DDCs*

Finally, an architectural modification to CHERI is considered to allow multiple *DDCs* to coexist. This approach has not been implemented. This would involve adding additional *DDC* registers, as illustrated in Figure 3.5. During compartment switches, the switcher sets one *DDC* to cover compartment bounds as before, while other *DDC* registers are set to cover shared memory, thus enabling multiple non-contiguous regions to be addressable at once and removing the need to perform *DDC* switching or use overlapping *DDC* bounds.

In terms of hardware cost, this approach would require additional *DDC* registers, along with related logic to determine the correct *DDC* capability to use.

This approach would also incur a low performance penalty since it requires no additional instrumentation or exceptions, and requires only annotating shared data as with other shared data approaches.

3.4 Summary

This chapter has detailed the requirements for a successful implementation of a compartmentalisation mechanism and the overview of the system to be implemented. Additionally, it has proposed five data sharing approaches, each representing a different point in the performance, engineering, and scalability trade-off space. These are manual capability annotations and shared data memory, with different approaches to accessing shared data from within a compartment. While the granularity of shared data varies, each provides strong guarantees of security, isolating a compartment to its own portion of the address space.

This chapter has addressed part of **RQ1**, examining which compartment models can be achieved in hybrid mode and the two most practical approaches have been

selected for evaluation based on the stated requirements: the overlapping shared data region approach, since this implements shared data with the least amount of engineering effort and performance penalty, at the cost of scalability, and manual capability propagation. The latter will be used for small scale compartments since it is not practical to apply to larger code portions, but provides low performance overhead and tightly bounded data sharing.

Chapter 4

Implementation

This chapter first details the porting effort of FlexOS to enable execution in hybrid mode on Morello. Next, the implementation of the CHERI backend, including the switcher and gates are described, followed by the implementation of data sharing approaches and the programming annotations and abstractions needed to enable them. This chapter continues to address **RQ1**.

4.1 Porting FlexOS To Morello In Hybrid Mode

Implementing a CHERI compartmentalisation backend first required a hybrid mode port of FlexOS. Unikraft upon which FlexOS is based, supported AArch64 already, meaning that existing AArch64 components could be adapted. The main steps taken to enable capability support within FlexOS are detailed. The total porting effort, including implementing the compartmentalisation backend, involved 2200 lines of code.

4.1.1 Booting

Capability features are enabled by the initial boot code. This includes disabling the trapping of Morello capability features at all exception levels. Additionally, the behaviour of capability features such as sealing return capabilities is defined at this stage, and the capabilities needed to locate the exception vectors are initialised. Return capability sealing is disabled in this implementation.

Capability Memory Access Faulting

The Morello architecture controls access to capabilities in memory, with the ability to generate a capability fault upon attempts to load or store valid capabilities. This behaviour can be used to restrict the rights of the system, however this behaviour needs to be disabled for FlexOS to allow capabilities to be loaded from and stored to memory.

Store faulting can be enabled and disabled in the block and page descriptors at stages 1 and 2 of translation. Specifically, 2 registers: TCR_ELx for stage 1

translation and VTCR_EL2 for stage 2, x corresponds to the relevant exception level of the processor (EL). If stage 2 translation is disabled, then the latter register is not relevant. Table 4.1 below lists the registers that need to be considered for capability store faulting.

Table 4.1. Bits controlling capability store faulting.

Translation stage	Register	Bit	Purpose	Value
1	TCR_ELx	CDBM (bit 59)	Enables or disables tracking stores of valid capabilities, `0b1` enables tracking, `0b0` makes the register have no effect.	`0b0`
1	TCR_ELx	SC (bit 60)	`0b0`: when CDBM is 0, fault, otherwise no effect. `0b1`: has no effect.	`0b1`
1	TCR_ELx	HPD0 (bit 41) HPD1 (bit 42)	HPD0 affects TTBR0_EL1 and HPD1 affects TTBR1_EL1. These need to be set to 1.	`0b1`
1	TCR_ELx	HWU060 (bit 44) and HWU160 (bit 48)	Affects TTBR0_EL1 and TTBR1_EL1 respectively. These need to be set to 1 depending on the translation table used.	`0b1`
2	VTCR_EL2	CDBM (bit 59)	Enables or disables tracking stores of valid capabilities, `0b1` enables tracking, `0b0` makes the register have no effect.	`0b0`
2	VTCR_EL2	SC (bit 60)	`0b0`: when CDBM is 0, fault, otherwise no effect. `0b1`: has no effect.	`0b1`
2	VTCR_EL2	HWU60 (bit 26)	Needs to be set to 1.	`0b1`

Load faulting similarly prevents the loading of a capability by generating a capability fault to prevent unauthorised access. Table 4.2 lists the registers and bits needed to disable faulting. Once again, if stage 2 translation is disabled, then VTCR_EL2 has no effect.

4.1.2 Exceptions

Handling exceptions in a compartmentalised system requires the use of banked registers. Through the use of executive and restricted modes, different views of the *DDC* are available to the system based on current execution mode. In order to give the exception handler adequate bounds and permissions in hybrid mode to perform exception handling, it uses the default capability in the *DDC*, which covers the entire address space.

Table 4.2. Bits controlling capability load faulting.

Translation stage	Register	Bit	Purpose	Value
1	TCR_ELx	LC (bits 61 and 62)	`0b00`: will zero capability tags, `0b01`: will have no effect, `0b10`: if CCTLR_ELx.TGENy is 1, fault loads of valid capabilities otherwise no effect, `0b11`: if CCTLR_ELx.TGENy is 0, fault loads of valid capabilities otherwise no effect. 'x' and 'y' refers to the translation table, i.e. TTBRy_ELx.	`0b01`
2	VTCR_EL2	LC (bit 61)	`0b00`: zero capability tags. `0b01`: has no effect.	`0b01`
1	CCTLR_ELx	TGEN0 and/or TGEN1	Apply to TTBR0_ELx and TTBR1_ELx. `0b0`: fault when TCR_ELx.LC is `0b11`, `0b1`: fault when TCR_ELx.LC is `0b10`. Only necessary if TCR_ELx.LC has been set to one of the values	`0b0`

All compartments operate in the so-called restricted mode. This grants them a view of the system registers which contain the restricted compartment capabilities. Upon taking an exception, the exception handler capability is used, which switches execution to executive mode. In executive mode, a different bank of registers is visible to the system, and these contain capabilities with wider bounds and greater permissions.

The exception handler is modified from a standard AArch64 implementation in two main ways. Firstly, capability registers are pushed to the stack to preserve any capabilities. Additionally, new capability exceptions such as a capability bounds fault are added to the handled exceptions.

4.1.3 Allocators

During the boot process, each compartment is assigned a dynamic memory allocator, which is initialised in the compartment heap, which is a region of memory defined statically in the linker script at build time by the developer, and is used during execution for dynamic memory allocation. All allocators are initialised at boot time in their respective heaps. Each compartment can only access allocators for memory which its *DDC* bounds cover. When allocating memory, the correct allocator is selected and returned by reading the compartment ID register.

4.1.4 UART

Universal asynchronous receiver/transmitter (UART) is a protocol used to exchange serial data between two devices. Access to UART is needed to output data from the system and is accessed by reading from and writing to a defined

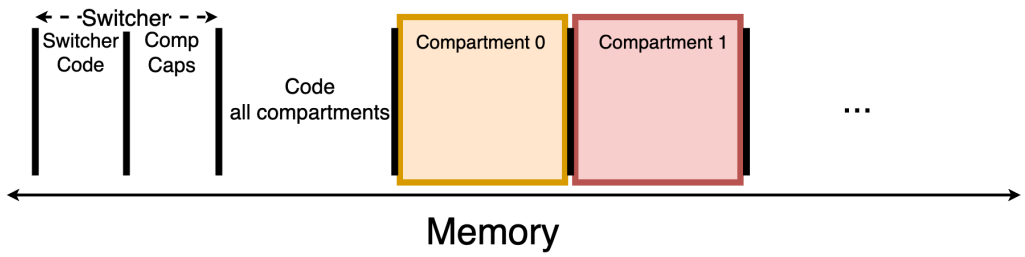

Figure 4.1. Memory layout of FlexOS on Morello.

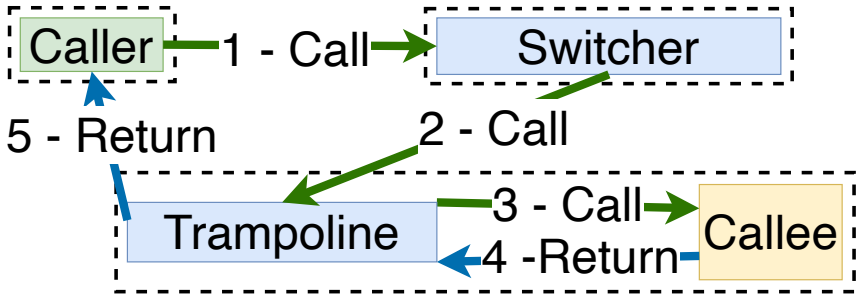

Figure 4.2. Control flow of a compartment switch (call and return paths). Dashed boxes represent compartments. The trampoline is available in the compartment of the callee compartment.

set of memory addresses. Default capability bounds allow access to UART since these cover the entire address space, however once a compartment is entered, UART is no longer accessible because the memory relating to UART is outside of the bounds of any compartment. Capabilities are initialised at boot which grant bounded access to UART addresses, to enable applications to print via serial. Functions which access UART addresses are ported to use the new UART capabilities.

4.2 Isolation Mechanism Implementation

The main components necessary to implement CHERI compartmentalisation in hybrid mode on Morello with FlexOS are now described, beginning with an overview of the system from a high level, then delving deeper, first into the structure of a compartment, including how isolation between compartments is enforced at the hardware level. Further, the initialisation of compartments is detailed, as well as the components of the switching mechanism: the gate, the switcher and the trampoline. Finally, the annotations required to restore limited data sharing between compartments are described.

4.2.1 CHERI Isolation Mechanism Overview

Global compartment capabilities are initialised at boot time on the compartment boundaries defined at build time in the linker script. These capabilities are

known as the *Default Data Capability (DDC)*, which restricts data accesses, and the *Program Counter Capability (PCC)* which restricts code execution. Once the developer has defined their compartment scheme, gates are inserted in place of function calls by the build tool, where these calls now cross compartment boundaries. The flow of execution is visualised in Figure 4.2. Gates are the beginning of the switching process, which involves storing the data needed for a compartment to resume execution on the compartment stack, as well as loading the data needed by the switcher and arguments required by the function call. The gate then loads and unseals a capability granting access to a capability pair for the switcher and invokes the switcher (step 1 of Figure 4.2). The switcher then performs the switching, including switching the stack and setting the new compartment *PCC* and *DDC*. It then calls a trampoline function (step 2), which calls the desired function (step 3). On completion, the callee function returns to the trampoline (step 4) and the trampoline performs the compartment switch back to the caller (step 5), where the caller context is then restored by the call gate. The trampoline acts as the entry and exit point to a compartment.

4.2.2 Compartment Structure

Compartments are defined at build time in a configuration file provided to the FlexOS build tool by the developer. This generates a separate region in the global linker script per compartment and a local linker script for libraries. At link time, isolated data are then placed into their respective, physically separate portion of the executable. Non-isolated data are placed into a default compartment. This process places all compartment static data into its respective compartment. If a shared data region model is used, data must be specifically shared via an annotation applied by the developer. The boot code uses boundaries defined in the global linker script to define compartments. Compartment code remains in the default code area, meaning that no isolation is implemented between functions. Within each compartment linker section, the developer statically reserves space. This space serves as both the private compartment stack and heap.

The compartment switcher is located in separate memory from all other code. The switcher memory is outside the bounds of any compartment. This is done to control access to the switcher. In addition, compartment capability pairs (one *DDC* and *PCC* pair per compartment) are stored by the boot code in memory, which is not accessible from any compartment. This enables secure compartmentalisation, since a compartment cannot arbitrarily grant itself access to another compartment. This can be seen in Figure 4.1.

```
...
targets:
  - architecture: arm64
    platform: morello
compartments:
  - name: comp1
    mechanism:
      driver: morello
    default: true
  - name: comp2
    mechanism:
      driver: morello
  - name: comp3
    mechanism:
      driver: morello
libraries:
  uktime:
    is_core: true
    compartment: comp3
  vfscore:
    is_core: true
    compartment: comp2
  ramfs:
    is_core: true
    compartment: comp2
  tlsf:
    version: staging
    kconfig:
      - CONFIG_LIBTLSF=y
  pthread-embedded:
    version: staging
    compartment: comp1
  newlib:
    version: staging
    kconfig:
      - CONFIG_LIBNEWLIBC=y
    compartment: comp1
...
```

Listing 4.1. Example of developer supplied configuration.

4.2.3 Initialisation

Based on the compartment boundaries defined in the linker script, compartments are initialised at boot time. This means that core FlexOS boot code and platform code is trusted to initialise compartment capabilities correctly and securely. A compartment is defined by a *DDC* and a *PCC*. Since no isolation between code is present, the *PCC* of each compartment has the same bounds. The value (the target address) of each *PCC* is the trampoline function. The boot code sets compartment *DDC* bounds to cover the compartment memory, which was statically defined in the linker script. Capability bounds can only cover a contiguous portion of memory, meaning that all compartment data must be contiguous and separate from any other compartment, to avoid overlap. Once compartment capabilities have been created, they are stored in the memory reserved for compartment capability pairs (the *DDC* and *PCC* capabilities used to define a compartment).

During boot time, the system also initialises a capability pair for the switcher. This pair grants access to the switcher code, and the compartment capability

pairs. To prevent unauthorised execution of the switcher, the capability pair granting access to the switcher is also placed in memory which is out of bounds of any compartment. To access this pair, the boot code gives each compartment a sealed capability in its private memory during the initialisation process, which can be unsealed using a `lpb` (a load pair and branch) instruction. By sealing the capability, the switcher can only be accessed via the specified entry points. Finally, each compartment receives a private allocator, which is initialised by the boot code on the heap which was reserved for the compartment by the developer in the linker script. Using this allocator, the private stack for each compartment is initialised within the compartment heap.

At the end of the boot process, the capability pair for the default compartment is loaded and execution then enters the default compartment. Entering the default compartment for the first time is not done via the trampoline, so a temporary capability is used which points directly to the main function.

4.2.4 Compartment ID

During compartment switches, the processor's compartment ID register is modified to contain the ID of the callee compartment during the switch. The caller compartment ID is restored on return. The compartment ID register is a system register containing a capability which identifies the currently running context of a compartment. Modification is done via a system register instruction (`MSR`). The compartment ID capability is used to select the correct allocator in FlexOS.

4.2.5 Switching

Compartment switch gates are implemented as C macros. This allows compartment switch instructions to be directly inlined at the call site, avoiding the need for a function call to invoke the switcher. Unlike in other implementations of FlexOS call gates, such as MPK [15], the compartment switch gate does not perform the compartment switch, this is done to prevent compartments from accessing the capability pairs of other compartments. Instead, it loads the parameters needed by the switcher and invokes the switcher, delegating the switching to the isolated switcher. When initiating a switch, the compartment switch gate takes the caller and callee compartment IDs, the callee function pointer, a return variable pointer (if needed) and arguments to be passed.

The compartment switch gates follow the AArch64 calling convention for argument registers. Registers 0-7 can be used to pass arguments, which enables 8 arguments as in a regular function call. However, in a departure from the standard calling convention, arguments which are passed on the stack, and indirect return values via `x8` are not supported in this implementation.

The procedure used to invoke the compartment switcher is as follows: modified registers are pushed to the stack, the current `sp` and `fp` are saved, the switcher parameters are loaded and finally, the sealed capability granting access to the switcher capabilities is loaded, unsealed and the switcher is invoked using a `lpb` instruction. Once the switcher has been invoked, the PCC is restricted to only execute switcher code.

The switcher is an isolated entity which is trusted to perform the compartment switches. The switcher *PCC* is only able to execute switcher code, which is physically separate from all other code. The switcher *DDC* is the only way to access compartment capability pairs. Once the switcher is invoked, the following steps are taken:

1. Upon first entering the switcher, the caller compartment *DDC* is still in place. This, along with the return capability generated by the call to the switcher, is stored on the caller compartment stack. A sealed capability is generated which grants access to this stored capability pair.
2. The *DDC* is changed to the switcher *DDC*
3. Callee compartment capabilities are loaded
4. The callee compartment ID is set
5. Callee compartment *DDC* is set
6. The new `sp` and `fp` are loaded and set
7. The callee compartment *PCC* is used to leave the switcher and jump to the trampoline

The trampoline serves as both the entry and exit point for a compartment. Return to the callee compartment can only be performed via the capability pair stored on the caller stack, accessed via a sealed capability. Capability unaware code is unable to do this, so a trampoline is employed to achieve this functionality. The trampoline first stores the sealed capability created by the switcher, on the callee stack, then calls the target function. The link register now contains the return address of the trampoline as a regular 64-bit pointer. Upon, the trampoline pops the sealed capability from the stack, unseals the capability and performs a capability return to the caller compartment. The is done in the form of a `lpb` instruction, where the caller return *PCC* and the caller *DDC* are loaded, the return is performed, and the caller restores its *DDC* capability.

Upon return from the callee compartment, the gate also restores the `sp` and `fp` and restores any saved registers. If the function call returned a value, the gate will store the returned value in a provided variable pointer.

```
1 static void
2 store32_le(uint8_t dst[4], uint32_t w)
3 {
4   uint8_t *__capability dstc = cheri_setbounds((uint8_t *__capability)dst, sizeof(dst));
5   __flexos_gate(store32_le_morello, dstc, w, compartment1);
6 }
```

Listing 4. Example wrapper function needed to call a manually annotated function.

4.3 Data Sharing Methods

For the full evaluation, overlapping *DDC* shared data compartments have been implemented using mutual distrust, and manual capability propagation has been implemented to sandbox individual functions. The following section describes how these two data sharing methods are implemented and how proof of concept implementations for exception and load/store macro based shared data access are implemented.

4.3.1 Manual Capability Propagation

Manual capability propagation involves identifying sensitive functions and modifying them to replace all pointers with capabilities. To avoid having to rewrite all calls to the modified function, the developer creates a wrapper function which accepts pointers as before and converts the pointers to bounded capabilities before calling the new sandboxed function. The wrapper function uses the existing function signature. The sandboxed function is a renamed copy of the original function, which is then modified by the developer to use capabilities. The wrapper is shown in Listing 4 where the function is first called before pointer arguments are converted to bounded capabilities and the new sandboxed function is called. Sandboxing is used as a way to avoid having to rewrite extensive parts of the application source code, as would be the case if safeboxing were implemented. Instead, limited changes to the function itself are needed.

4.3.2 Overlapping Shared Data Region

Here, two compartments are contiguous in memory, interrupted only by a shared data region. The shared data region acts in the same way as each compartments' private heap, which is statically defined in the linker script. The developer defines shared data memory statically in the linker script, and it is then initialised during the boot process. A shared data allocator is also initialised during boot. In this model, each compartment retains its own private stack, heap, and allocator. Static shared data is identified by compiler annotations which were inserted by the developer, as shown in Listing 5. Shared data, which is dynamically allocated (including shared stack data), uses the shared allocator as shown in Listing 6.

```
1 struct Config config; //private
2 struct Config config __section(".data_shared"); //shared
```

Listing 5. Example of compiler annotation needed to share static variable needed for shared data approaches.

```
1 var = uk_calloc(comp1_allocator, 1, sizeof(struct var)); //private
2 var = uk_calloc(flexos_shared_alloc, 1, sizeof(struct var)); //shared
```

Listing 6. Example of dynamically allocated shared data for shared data approaches.

Special handling is required for shared string literals. String literals are by default placed in private memory, for this reason, shared string literals are wrapped in a macro to force relocation to shared data.

4.3.3 Exception-Based Shared Data Access

This solution requires a shared data capability to be retained in a register to enable fast swapping. To do this, the compiler is instructed by the developer to reserve a register, `c18` for the implemented proof of concept, since this register is often reserved for OS use anyway in the calling convention. The capabilities for shared data access are initialised during the boot process and are stored in the same reserved memory used to store compartment capabilities.

During a compartment switch, the switcher installs the correct shared data capability in `c18`. When a compartment attempts to access shared data, an exception is triggered and a special capability bounds fault exception handler checks the fault address register to ensure that the attempted access was within the bounds of shared data. It then switches the current *DDC* capability with the shared data capability. The compartment can now access shared data. When compartment private data is now accessed, the same procedure is followed, except that the handler checks that the attempted access was within the compartment *DDC* bounds.

The decision to permanently maintain the shared data DDC in a register is for performance reasons, avoiding the need to load it from memory each time. However, reserving a register can have a negative effect on performance, although this is minimised by the use of `c18` which is often reserved.

4.3.4 Macro-Based Shared Data Access

Similarly to exception-based shared data access, a register (`c18`) is reserved by the compiler for the shared data capability, which is initialised during boot and installed during compartment switching. However, rather than relying on exceptions to switch the *DDC* capabilities, shared data accesses occur directly via the capability. The developer wraps shared data in a C macro which performs the

```
1  var = *shared_data; //original access
2  MORELLO_LOAD_SHARED_DATA(shared_data, var); //wrapped shared data access
```

Listing 7. Example of macro based shared data access.

capability load or store and, in the case of a load, returns the loaded data in a register corresponding to a specified variable as shown in Listing 7. The macro places loaded data in a free register if possible, to avoid the need for an additional memory access. Where this is not possible, for example when multiple accesses occur in quick succession, loaded values may need to be pushed to the stack.

4.4 Summary

This chapter has described the porting which was necessary for FlexOS to leverage CHERI hybrid mode on Morello. Following this, the implementation of the CHERI backend was detailed, including the switcher, trampoline, and call gates. Finally, the implementation of data sharing mechanisms was described. This chapter addressed **RQ1**.

Chapter 5

Evaluation

In this chapter, different applications are compartmentalised using the implemented CHERI backend on FlexOS, and evaluated with overlapping shared memory and manual capability propagation. First, the engineering cost associated with SQLite, with two compartments isolated at the library level and five Libsodium functions which are sandboxed using manual capability propagation. Next, the performance implications of these approaches are evaluated, followed by an evaluation of the cost of a compartment switch, through the use of microbenchmarks, including the effect of microarchitechtural features such as cache latency on the switch latency as a whole. Evaluating these different configurations, highlights the options available within the performance and engineering effort trade-off space. Finally, the interface security properties achieved by the CHERI backend are compared both with shared data and capability annotations, with the other FlexOS backends: MPK and EPT.

This evaluation does not compare the implemented CHERI compartmentalisation backend on FlexOS and associated data sharing methods with other CHERI works (Section 2.6.2) because these are implemented on other platforms (CHERI-MIPS and CHERI RISC-V) for which hardware was not available. Further, many existing CHERI compartmentalisation works target pure capability mode, which is out of scope for this exploration and evaluation of hybrid mode compartmentalisation within a single address space.

This chapter focuses on **RQ1**, **RQ2** and **RQ3**: the performance, engineering and security implications of compartmentalisation using the CHERI isolation mechanism are evaluated and compared to MPK and EPT.

5.1 Evaluation Setup

All evaluation is conducted using a Morello board, running a Morello SoC [186]. FlexOS is executed running bare metal on the hardware. The results displayed are an average of 10 runs. Data gathered from Linux on Morello is as a result of running under a minimal, capability-unaware, AArch64 Debian 11 installation. The Morello CPU is clocked at 2.5GHz throughout for all experiments, and precise

timing measurements are taken by the generic 50MHz system clock. The plots shown do not have error bars because the data collected is stable, making it unnecessary.

The libsodium library is integrated with a benchmark which is derived from its test suite, running representative tests (e.g. encrypting a buffer, generating a key) 200 times in a loop. The compartmentalisation scenario and the benchmark for SQLite are both taken from the FlexOS paper [15]. The baseline used for all comparisons is FlexOS with no compartments or isolation, running the benchmark.

5.1.1 Rationale For Compartmentalised Components

Below, the compartmentalisation of components is motivated in both the mutual distrust shared data model and function sandboxing model.

SQLite

`vfscore` & `ramfs` Together, these two libraries represent a high value target for compromise, since the filesystem often has access to sensitive data and critical system resources. In addition, compartmentalising the filesystem is roughly similar to userland execution in Linux where the filesystem is part of the kernel, where the effective protection compartment switch is a system call. Here, mutual distrust is used.

Libsodium

`sodium_hex2bin` & `sodium_bin2hex` Both functions (`sodium_hex2bin` & `sodium_bin2hex`) handle potentially risky input strings and write output to buffers. This could be exploited by a malicious actor to perform buffer overflow attacks on the rest of the system. To prevent this, both functions are sandboxed.

`chacha20_encrypt_bytes` Encrypt bytes performs encryption on untrusted input buffers, which can potentially be exploited by external user input. Sandboxing is used to constrain execution.

`store32_le` & `store64_be` Both functions (`store32_le` & `store64_be`) store data in memory. While the functions themselves are simple, they serve as a proof of concept implementation for frequently used, high value targets for exploitation, since they are called from numerous places and store data in memory. For this reason, both functions are sandboxed.

Table 5.1. Porting effort required to compartmentalise.

Software	Sharing approach	Compartments	Porting cost	Changes (LoC)
libsodium	Function sandboxing	`sodium_hex2bin`	< 1h	9
		`sodium_bin2hex`	< 1h	8
		`chacha20_encrypt_bytes`	< 2h	73
		`store32_le`	< 1h	5
		`store64_be`	< 1h	5
SQLite	Overlapping DDCs	`vfscore + ramfs`	< 2d	< 300

5.2 Engineering Cost

Below, the engineering cost associated with the two compartmentalisation models is considered.

5.2.1 SQLite

Table 5.1 shows the porting effort associated with the compartmentalisation of SQLite on FlexOS. The configuration chosen (`vfscore` and `ramfs`) can be achieved by an experienced engineer in under 2 days. The effort involved can be broken down into two main sections, 1) gate insertion and 2) data relocation. Gate insertion is mostly automated by the FlexOS toolchain, with the programmer only needing to insert annotations at the desired compartment boundary. The majority of the work comes from relocating data. The data which must be relocated is shared data. Data is shared via a shared data region of memory. Static data such as strings must, therefore, be manually annotated by the developer to place them in the shared data section if they need to be accessible outside the compartment. In the same vein, data allocations must be annotated by the developer to now allocate in the shared data section with a shared allocator.

The engineering cost is higher than that associated with sandboxing individual functions. However, the isolation achieved is also greater than an individual function. It instead covers the entire filesystem, with the entire compartment covering 5.8k lines of code. The number of annotations required for porting (<300 out of a total code base of >900k lines of code) represents 0.0003% of the code base, which is within the requirements of $< 0.2\%$ for the application level. The time to port is between less than the 1 week allowed for the OS level and greater than the 1 day allowed for application level.

5.2.2 Libsodium

The engineering cost of isolating functions is lower per function than it is for compartmentalising `vfscore` and `ramfs` for SQLite (Table 5.1). However, the isolation

achieved is less sweeping and more granular, and the trust model is different. Here, a function is sandboxed. Relatively small functions have been chosen to reduce the complexity of porting, with the largest function being `chacha20_encrypt_bytes` with 141 lines of code, requiring 73 changes, which represents >50% of the function. The type of engineering effort is also different. Whereas to isolate a library, the effort mostly involves annotations, here an understanding of CheriC and capabilities is required for successful isolation; the function must be modified by the developer to accept capabilities in place of pointers. It is also easier to make mistakes when compartmentalising. This is because the sandboxed function is granted access to shared data via a capability. If bounds are incorrectly set on this capability, access may be granted inadvertently to compartment private data.

The amount of code to port, to sandbox functions, can be lower than that needed for SQLite and so is also within the requirements (Section 3.1). The time to port, while lower in absolute time, is higher when the scope of isolation is considered (filesystem versus a single function). However, this is also within the required time (Section 3.1). Attempting to port `vfscore` and `ramfs` in this way would have presented an engineering challenge due to the need to propagate capabilities through a much larger part of the code base.

5.2.3 Summary

Overall, sharing data between compartments in hybrid mode presents challenges. Attempting to propagate capabilities through larger parts of the code base than individual functions can result in a high engineering burden. This means that compromises must be made to reduce the engineering effort, which may result in compromises being made in the design of a system. Overlapping shared data, applied to SQLite, presents a lower engineering effort when applied to larger parts of the code base such as the filesystem, but comes at the cost of coarser data sharing, and it is potentially difficult to apply to more than two compartments.

5.3 Performance

The performance cost of isolating the filesystem from SQLite, and individual Libsodium functions is evaluated.

Table 5.2. SQLite 2 compartments (vfscore + ramfs isolated) compartment switch metrics.

0 to 1	1 to 0	Total	Switches/1k Instructions
185066	585271	770337	2.49

Table 5.3. Performance counters by for each configuration compared to the uncompartmentalised baseline.

Configuration	L1I Acc	L1I Miss	L1D Acc	L1D Miss	Br Ret	Br Mispred	Mem Acc	Inst Ret
`sodium_hex2bin` & `sodium_bin2hex`	+0.15%	+3.95%	+0.09%	+25.11%	+0.29%	+10.25%	+0.13%	+1.74%
`chacha20_encrypt_bytes`	+23.58%	+209.49%	+16.85%	+106.00%	+23.33%	+76.76%	+16.69%	+4.90%
`store32_le` & `store64_be`	+16.28%	-37.47%	+13.98%	+12.03%	+14.29%	+91.72%	+13.84%	+2.90%
`libsodium all`	+16.36%	+178.92%	+12.44%	+189.79%	+15.73%	+102.34%	+12.15%	+4.54%
`SQLite`	+99.6%	+46.0%	+48.1%	+18.4%	+6.5%	+175.2%	+48.3%	+27.1%

Table 5.4. Libsodium configurations compartment switch metrics.

Configuration	0 to 1	1 to 0	Total	Switches/1k Instructions
`sodium_hex2bin` & `sodium_bin2hex`	33600	0	33600	0.005
`chacha20_encrypt_bytes`	93991	4408608	4502599	0.669
`store32_le` & `store64_be`	2874836	0	2874836	0.435
`all`	791041	2206888	2997929	0.447

5.3.1 SQLite

First, the data obtained from the evaluation of SQLite is presented, which has been compartmentalised into two compartments, with `vfscore` and `ramfs` isolated from the rest of the system. In this scenario, isolation is at the library level, with compartments mutually distrusting. A benchmark which performs 5000 `INSERT` operations on an in memory (`ramfs`) database was used. SQLite was selected because it features a large number of system calls and is a widely used application. It was also used to evaluate compartmentalisation with MPK and EPT on FlexOS running on x86 [15]. In the compartmentalisation scenario selected, the filesystem is isolated. Consequently, this benchmark also features a large number of compartment transitions (2.49/1k Instructions), as shown in Table 5.2.

CHERI3 (CHERI with 3 compartments) is implemented by combining the overlapping *DDC* shared data approach with an additional compartment isolating *uktime* using manual capability propagation, which is possible as *uktime* is a very small library. This done to aid comparison with the x86 scenarios presented in the FlexOS paper [15], but in practice adds negligible overhead since *uktime* is called at most 3 times.

The baseline used is uncompartmentalised FlexOS, which represents a standard unikernel without isolation. Isolating the filesystem adds an overhead of 119.9% (Figure 5.1). This translates to a runtime of 0.113s compared to 0.051s for the baseline (Figure 5.2). This slowdown can be attributed to the fact that the isolated libraries lie on the hot path, meaning that they are frequently called. However, the runtime of the compartmentalised system still outperforms the same benchmark running on an unmodified Linux installation (0.158s vs 0.113s compartmentalised).

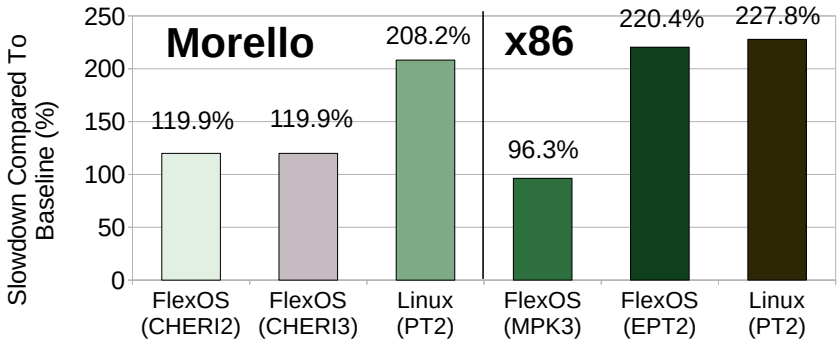

Figure 5.1. Overhead relative to uncompartmentalised FlexOS on respective system (Morello and x86), of SQLite configurations. Uncompartmentalised FlexOS is used as a baseline because this represents a standard unikernel lacking isolation between the application and kernel. CHERI3 included with `uktime` in compartment 3 (manual capability propagation for comparison with MPK3). Also shown are the overheads of MPK3, EPT2 and Linux taken from [15].

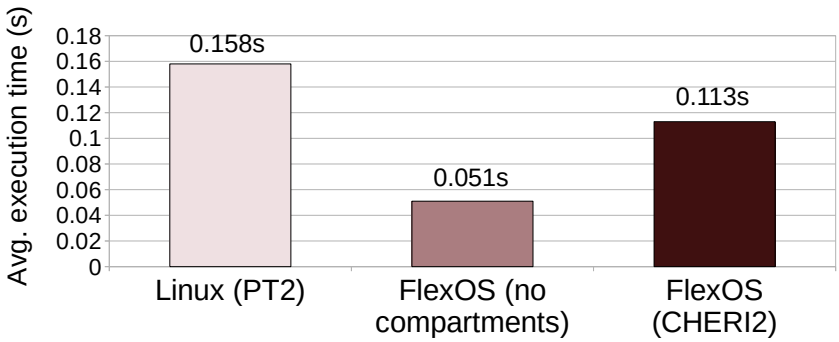

Figure 5.2. Execution time in seconds of different configurations of SQLite running on Morello.

Running on Linux is roughly equivalent to a two compartment scenario, due to the user-kernel separation at the filesystem boundary. This shows adding the hybrid mode CHERI compartmentalisation backend preserves the performance advantage of running the benchmark on a unikernel. The relative overheads are also in line with the relative overheads achieved by other FlexOS isolation mechanisms, shown in (Figure 5.1). Compared to the overhead of MPK (MPK3), CHERI only is slightly more expensive. This can be attributed to the switching mechanism, which using MPK needs only a gate, but with CHERI performs jumps to a switcher and to a trampoline, as well as utilising a gate. The CHERI backend outperforms EPT (EPT2).

Performance counter data, displayed in Table 5.3, was collected for SQLite when running under the baseline and when compartmentalised. It shows that compartmentalisation increases the number of instructions executed by 27.1% and memory accesses by 48.3%. Correspondingly, the number of L1 instruction cache and L1 data cache accesses increases, while the number of misses for both increases by a smaller proportion than the increase in accesses. Interestingly, the branch mis-prediction rate rises by a far greater amount than the number of branches executed. This may be attributed to the increased number of indirect branches

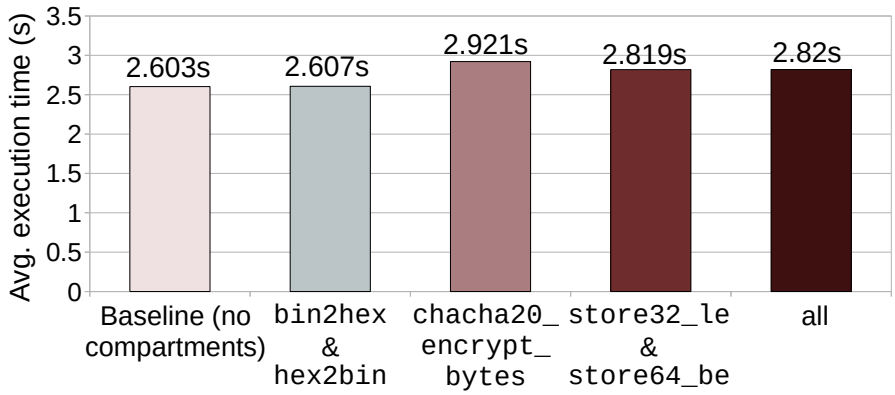

Figure 5.3. Execution time in seconds of libsodium configurations running on Morello.

used as a result of the switching process, which are harder for the predictor to predict.

Overall, evaluated against the DARPA CPM requirements detailed in Section 3.1, the performance cost of this approach is higher than that allowed, however, the performance is still better than running as a Linux application, similar to compartmentalisation implemented on FlexOS with MPK and faster than compartmentalisation implemented on FlexOS with EPT.

5.3.2 Libsodium

Different configurations of 5 `Libsodium` functions which have been sandboxed are analysed. Here, the functions are not trusted and are isolated from the rest of the application, which has access to their private data. A workload which runs a number of `Libsodium` tests 200 times in a loop was used. The performance overhead added by isolating the selected `Libsodium` functions is much lower than isolating the filesystem when running SQLite. This is due to fewer compartment switches (Table 5.4); the highest is `chacha20_encrypt_bytes` with 0.669 compartment switches/1k instructions. The lowest performance overhead is achieved when `sodium_hex2bin` and `sodium_bin2hex` are isolated, adding only a 0.144% performance overhead. In contrast, the highest performance overhead comes from compartmentalising `chacha20 _encrypt_bytes` only, with an overhead of 12.207%. This is higher than the scenario where all are isolated, because `chacha20_encrypt_bytes` makes calls to `store32_le`. When only `chacha20_encrypt_bytes` is isolated, a compartment switch is required for each call, hence the overhead is higher. Evaluating against the DARPA CPM requirements for function granularity isolation [27], these results are in range of the required 5% overhead, whilst providing the formally verified security guarantees of CHERI and preventing oversharing of arguments. It also shows that the choice of function to isolate is also an important consideration.

These results carry over to the performance counter data shown in Table 5.3. The number of instructions executed, memory accesses performed including cache ac-

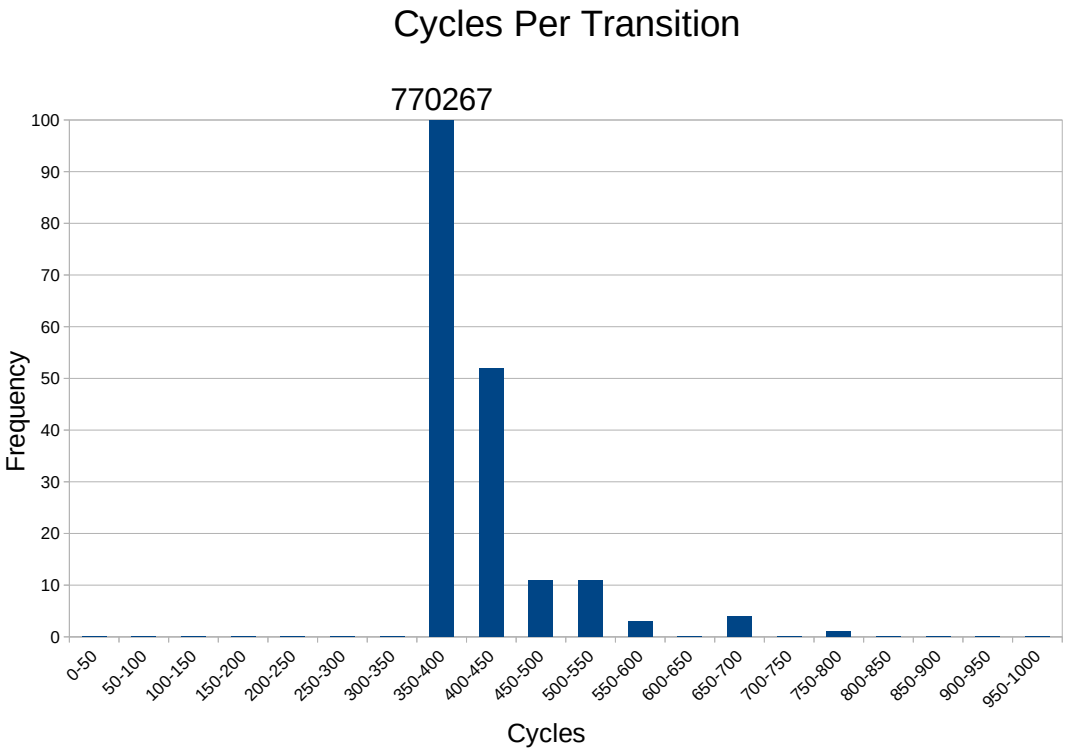

Figure 5.4. Number of cycles per compartment switch (SQLite CHERI3)

cesses and misses and branches executed, all rise in proportion. As with SQLite, the branch predictor struggles with increased use of indirect branches. The reduction in instruction cache misses observed with `store32_le` & `store64_be` is because the new wrapper function is no longer inlined as the original functions were, resulting in more efficient cache utilisation.

5.3.3 Microbenchmarks

It is important to consider the actual cost of switching between compartments, since this is often where much of the overhead comes from [208]. To this end, microbenchmarks are used to obtain a breakdown of the cost in CPU cycles associated with a switch. This is illustrated in Figure 5.5. Compartment switches can be broken down into hot and cold switches, where hot switches are ones which take <400 cycles. This is a result of cache utilisation. The vast majority of switches (>99.9%) (Figure 5.4) observed in all configurations fall into the category of hot switches. The cold switches represent a worst-case cycle latency for compartment switches. These can be expected in compartmentalisation scenarios where compartment switches occur rarely and cache utilisation is worse.

When looking at the components of a switch, it can be seen that a large proportion of the switching latency is due to the switcher and trampoline. The branches from the gate to the switcher and from the trampoline to the function depend heavily on cache utilisation to be efficient. This results in a best switching latency of <400 cycles and a worst case of 900-1000 cycles.

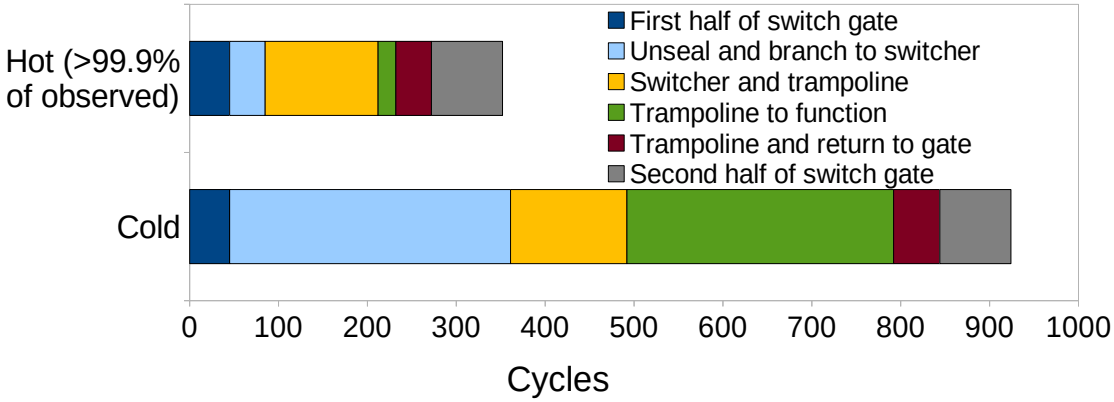

Figure 5.5. Hot and cold compartment switch latencies, broken down into component parts.

5.4 Interface Security Properties

The following section considers and compares the interface security properties offered by both the overlapping shared data region approach implemented with SQLite (SD) and the manual propagation of capabilities used to sandbox libsodium functions (SF), compared to the FlexOS MPK and EPT backends. In order to conduct this portion of the evaluation, the 8 compartment interface vulnerabilities defined by Lefeuvre *et al.* [28] will be used. This section considers the protection offered by these scenarios without modifying or sanitising the compartment interfaces in any way.

5.4.1 Exposure of Addresses

Addresses which are internal to compartments can be exposed for a number of reasons, including uninitialised data structures. All of the considered isolation mechanisms are vulnerable. In the case of SF, it is vulnerable by the very nature of the way it shares data: capabilities contain as their value the data address. A way to address this would be to use a technique such as pointer swizzling, as used by RLBox [10].

5.4.2 Exposure of Compartment-Confidential Data

All models are vulnerable to exposure of compartment confidential data. In the case of SD, MPK and EPT, shared data is used where the potential for oversharing exists. In addition, uninitialised objects can affect all mechanisms where care is not taken to avoid this. The risk of exposing confidential data can be reduced by implementing checks and analysis to ensure that confidential data does not cross compartment boundaries [16] and uninitialised memory is not used.

5.4.3 Dereference of Corrupted Pointer

Corrupted pointers can affect all mechanisms. In the case of SD, MPK and EPT, if an attacking compartment feeds corrupted pointers to a victim compartment, it could cause either the wrong shared data to be accessed or an denial of service. SF can be affected if the sandbox returns a corrupted pointer which is dereferenced without being sanitised. However, such attacks are also easily mitigated by implementing checks to verify an expected address range of a pointer.

5.4.4 Usage of Corrupted Indexing Information

SD, SF, MPK and EPT are all susceptible to such an attack, careful checks must be implemented to mitigate this. While the arguments passed into the sandbox cannot overflow the capability bounds, corrupted indexing information could still be used to perform a denial-of-service attack by attempting to access outside these bounds. Additionally, a sandbox could be tasked with returning indexing information to be used by unsandboxed code. Here, care must be taken to sanitise such values to ensure that an expected value is used.

5.4.5 Usage of Corrupted Object

Similarly, all mechanisms are vulnerable to using a corrupted data object. SD, MPK and EPT may access shared data which has been attacked. SF may use a data object which a sandbox was granted access to and corrupted, or one which the sandbox returned and was corrupted. In all cases, checks must be implemented.

5.4.6 Expectation of API Usage Ordering

No guarantees are made by any mechanism about the usage ordering of API methods. Detecting incorrect API ordering is a challenging task, especially where there is no clear specification. Tools such as APISan [181] and ARBITRAR [209] can help to detect API misuses, but cannot detect all such uses.

5.4.7 Usage of Corrupted Synchronisation Primitive

No guarantees are made by any mechanism about synchronisation primitives. Addressing this class of vulnerabilities requires careful reworking of multi-threading in applications to ensure that proper checks are in place.

5.4.8 Shared-Memory Time-of-Check-to-Time-of-Use

SD, MPK and EPT are all vulnerable to this type of attack. SF is less susceptible to being attacked in this way since functions tend to perform a single task and so there is less opportunity to corrupt data between the check and use, however it is still vulnerable. Addressing this can be done by working only on copies of data [10] or preventing concurrent access to data. Both can result in a reduction in performance.

5.4.9 Summary Of Security Properties

Concretely, the two compartment models, SD and SF implemented using CHERI do not improve upon the guarantees offered by the MPK and EPT mechanisms. The security on offer depends heavily on the precise nature of the implementation and how well data flows between compartments are sanitised.

5.5 Summary

This chapter presented an evaluation of different compartmentalisation scenarios using the implemented CHERI backend on FlexOS and data sharing mechanisms. The evaluation was conducted using two popular applications. SQLite was compartmentalised using overlapping shared memory, with compartments mutually distrusting. Libsodium functions were sandboxed by manually annotating code to use capabilities. The engineering effort, performance implications and interface security properties were then evaluated. The evaluation revealed that keeping the engineering cost low when implementing hybrid mode compartmentalisation in a single address space unikernel, requires compromises to be made to the size of the code being isolated or the security of data sharing. However, the performance cost of the CHERI backend is comparable to MPK and outperforms EPT for the same SQLite scenario on FlexOS, and within the DARPA requirements [27] when individual functions are sandboxed. Additionally, FlexOS with the implemented CHERI backend outperforms the same benchmark on a Linux system, where both FlexOS and Linux are isolating the filesystem. Finally, the interface security properties offered by the compartmentalisation scenarios are not improved when compared to MPK or EPT without additional interface sanitising.

This chapter addressed **RQ1**, **RQ2** and **RQ3**: the performance, engineering and security implications of compartmentalisation using the CHERI backend compared to MPK and EPT.

Chapter 6

Conclusion & Future Work

The appeal of hybrid capability mode was the ability to implement compartmentalisation with minimal changes to complex or poorly maintained code bases, while still benefiting from the strong hardware enforced bounds implemented by the CHERI architecture.

This thesis set out to answer the following research questions:

RQ1 Which compartment models are possible using Morello, using what programming abstractions, at which refactoring costs, and how do they scale?

RQ2 How does Morello's compartmentalisation performance compare to other single address space compartmentalisation mechanisms, such as Intel Memory Protection Keys (MPK)?

RQ3 What security properties does CHERI/Morello-based compartmentalisation offer, versus mechanisms such as MPK?

Addressing **RQ1**: the design and implementation of hybrid mode compartmentalisation in FlexOS, using CHERI was presented. Compartmentalising applications represents a trade-off between the desire for maximum performance and throughput of a system, and the effort needed by the developer to implement compartmentalisation, often in retrospect, to an application and libraries. Achieving this with low performance overhead and engineering cost requires carefully considering how data will be shared between compartments. Five different techniques to achieve this were proposed, ranging from annotations, to manually rewriting software to use capabilities. FlexOS was ported to the Morello platform, running bare-metal on the hardware in hybrid mode. Following this, a compartmentalisation mechanism was proposed and implemented on top of FlexOS. SQLite and its filesystem were then partitioned into mutually distrusting compartments using a shared data approach, where data is shared via annotations and the compartment bounds overlap with the shared data memory. Next, sandboxed Libsodium functions were implemented, where pointers are annotated and transformed into capabilities, and code is rewritten to take advantage of capabilities.

Due to the flexibility afforded by CHERI, smaller portions of the system can be compartmentalised, meaning that depending on the performance budget avail-

able to the developer, light-weight isolation can be implemented in the form of sandboxes. Porting functions to use capabilities can result in a large amount of engineering effort, but for small functions this is reduced and offers strong sandbox guarantees with tightly bounded access to argument data. For reduced engineering effort when compartmentalising larger portions of code, it is possible to use an overlapping shared data approach between pairs of compartments, although here data sharing is coarser, providing weaker security, and this solution struggles to be applied to more than two compartments which limits its use.

In answering **RQ2**, the implemented system on Morello was evaluated. The CHERI backend is comparable in terms of performance overhead when compared to the previously implemented MPK backend (119.9% overhead vs 96.3% overhead), and the performance overhead is lower than the EPT backend (220.4%). FlexOS with CHERI compartmentalisation in hybrid mode still outperforms the same benchmark running on a standard Linux system on Morello hardware, meaning that the performance advantage of a unikernel has not been negated due to the additional isolation added. When functions are selected wisely, the performance overhead is as low as 0.144%, and within the <5% required by the DARPA CPM call for proposals [27]. The latency associated with compartment switches was also evaluated. These can add significant overhead due to the need to securely switch compartment capabilities and stacks.

Finally, **RQ3** was addressed in Section 5.4. Despite the strong hardware bounds and permissions checking enforced by CHERI, the implemented compartmentalisation suffers from the same interface vulnerabilities as MPK and EPT.

Overall, based on the performance figures collected, and the experience gained from compartmentalising applications using different methods, implementing compartmentalisation with CHERI in a single address space unikernel in hybrid mode, is challenging. Primarily, this stems from the need to share data between isolated compartments. When applications are fully ported to use capabilities, this can be done via capability argument passing, however, in hybrid mode, sharing data securely between capability-unaware compartments results in engineering effort requiring compromises to scalability and security to keep it low.

6.1 Contributions

The following are the contributions made during this research:

1. An investigation of hybrid mode compartmentalisation in a single address space unikernel, evaluated on Morello hardware. To achieve this:

    - FlexOS was ported to Morello in hybrid mode.

- A compartmentalisation mechanism utilising CHERI in hybrid mode was proposed.
- The compartmentalisation mechanism was then designed and implemented, to enable compartmentalisation with CHERI in hybrid mode, in a single address space, statically linked unikernel.
- Five data sharing methods for hybrid mode compartmentalisation were proposed.
- Data sharing methods for hybrid mode compartmentalisation were designed and implemented.

2. An evaluation of hybrid mode compartmentalisation and data sharing mechanisms, finding that:
   - The performance is in line with MPK and outperforms EPT on x86 when applied to compartmentalisation of a unikernel.
   - FlexOS with CHERI compartmentalisation applied, outperforms Linux with the same application.
   - The engineering cost can be reduced by making trade-offs.
3. An analysis of the security properties of the implemented hybrid mode CHERI compartmentalisation mechanism, finding that:
   - Compartment interface vulnerabilities are not mitigated through the use of CHERI, with the system vulnerable to the same classes as MPK and EPT.

6.2 Future Work

6.2.1 Compiler Propagated Capabilities

A limitation of compartmentalisation works, including this one, is the manual effort required to port applications. An ideal scenario using hybrid mode would see all pointers which cross compartment boundaries replaced with capabilities, eliminating the need for any shared data, which is coarse and prone to oversharing.

To achieve this with lower engineering costs, compiler passes are needed to track pointers which cross compartment boundaries and then automatically transform them and variables which make use of them into capabilities. This would ease the burden of porting and has the potential to truly unlock the promised potential of hybrid mode to enable highly compatible compartmentalisation.

To help direct the propagation of capabilities within FlexOS, existing compartmentalisation annotations marking call gates can be used by the toolchain to

insert compiler annotations which can be used to identify which function calls and thus which pointers are relevant to track, reducing the number which must be modified and to retain more unmodified code. Pointers and their dependencies will be identified in LLVM IR form, with the necessary modifications carried out during this stage. However, fully automating this process is not possible because some code will need to be modified to work with capabilities. Implementing a compile time feedback system such as RLBox [10] would be a feasible solution.

6.2.2 Pure Capability Compartments

This thesis has investigated the use of hybrid mode in single address space compartmentalisation due to the promise of low engineering costs achieved through compatibility. Future work will consider what pure capability compartments look like in this context and how they can be applied while needing the minimum amount of porting work. All data accesses will occur via capabilities and all data sharing will also occur via capabilities, eliminating the need for the hybrid mode data sharing approaches explored in this thesis.

Additionally, some of the overhead associated with compartment switches in hybrid mode may be eliminated in pure capability mode. For example, it may be possible to implement a system requiring only a small switcher and no trampoline.

Hybrid mode compartments use a single set of global capabilities to restrict a compartment, this is no longer possible in pure capability mode with all accesses performed via capabilities. Any accesses to shared data will also need to sanitise capabilities which are loaded, to ensure that these do not escape compartment bounds.

References

[1] J. ”Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. DOI: "10.1109/PROC.1975.9939" (cited on pp. 14, 17).

[2] P. A. Karger, “Limiting the damage potential of discretionary trojan horses,” in *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, IEEE Computer Society, 1987, pp. 32–37. DOI: 10.1109/SP.1987.10011. [Online]. Available: https://doi.org/10.1109/SP.1987.10011 (cited on p. 14).

[3] N. Provos, M. Friedl, and P. Honeyman, “Preventing privilege escalation,” in *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.: USENIX Association, Aug. 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/preventing-privilege-escalation (cited on pp. 14, 22, 34).

[4] R. N. M. Watson, J. Woodruff, P. G. Neumann, *et al.*, “CHERI: A hybrid capability-system architecture for scalable software compartmentalization,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 20–37. DOI: 10.1109/SP.2015.9. [Online]. Available: https://doi.org/10.1109/SP.2015.9 (cited on pp. 14, 15, 32, 40).

[5] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang, “Codejail: Application-transparent isolation of libraries with tight program interactions,” in *Proceedings of the 17th European Symposium on Research in Computer Security*, S. Foresti, M. Yung, and F. Martinelli, Eds., Springer Berlin Heidelberg, 2012, pp. 859–876, ISBN: 978-3-642-33167-1 (cited on p. 14).

[6] H. Almatary, M. Dodson, J. Clarke, *et al.*, “Compartos: CHERI compartmentalization for embedded systems,” *CoRR*, vol. abs/2206.02852, 2022. DOI: 10.48550/arXiv.2206.02852. arXiv: 2206.02852. [Online]. Available: https://doi.org/10.48550/arXiv.2206.02852 (cited on pp. 14, 39).

[7] V. A. Sartakov, L. Vilanova, D. Eyers, T. Shinagawa, and P. Pietzuch, "CAP-VMs: Capability-Based isolation and sharing in the cloud," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 597–612, ISBN: 978-1-939133-28-1. [Online]. Available: `https://www.usenix.org/conference/osdi22/presentation/sartakov` (cited on pp. 14, 40).

[8] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proceedings of the 28th USENIX Security Symposium*, ser. USENIX Security'19, USENIX Association, 2019, ISBN: 978-1-939133-06-9. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner` (cited on pp. 14, 34).

[9] M. Hedayati, S. Gravani, E. Johnson, *et al.*, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *Proceedings of the 2019 USENIX Annual Technical Conference*, ser. ATC'19, USENIX Association, 2019, ISBN: 978-1-939133-03-8. [Online]. Available: `https://www.usenix.org/conference/atc19/presentation/hedayati-hodor` (cited on pp. 14, 26, 33).

[10] S. Narayan, C. Disselkoen, T. Garfinkel, *et al.*, "Retrofitting fine grain isolation in the firefox renderer," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 699–716, ISBN: 978-1-939133-17-5. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity20/presentation/narayan` (cited on pp. 14, 20, 27, 32–34, 75, 77, 81).

[11] D. Schrammel, S. Weiser, S. Steinegger, *et al.*, "Donky: Domain keys – efficient in-process isolation for RISC-V and x86," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX Security'20, USENIX Association, 2020, ISBN: 978-1-939133-17-5. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel` (cited on pp. 14, 26).

[12] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'17, Association for Computing Machinery, 2017, ISBN: 9781450349468. DOI: `10.`

1145/3133956.3134066. [Online]. Available: https://doi.org/10.1145/3133956.3134066 (cited on pp. 14, 24).

[13] M. Bauer and C. Rossow, "Cali: Compiler-assisted library isolation," in *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, J. Cao, M. H. Au, Z. Lin, and M. Yung, Eds., ACM, 2021, pp. 550–564. DOI: 10.1145/3433210.3453111. [Online]. Available: https://doi.org/10.1145/3433210.3453111 (cited on pp. 14, 24, 33, 34).

[14] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "CubicleOS: A library OS with software componentisation for practical isolation," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'21, Association for Computing Machinery, 2021 (cited on pp. 14, 26, 52).

[15] H. Lefeuvre, V. Badoiu, A. Jung, *et al.*, "Flexos: Towards flexible OS isolation," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, 2022, pp. 467–482. DOI: 10.1145/3503222.3507759. [Online]. Available: https://doi.org/10.1145/3503222.3507759 (cited on pp. 14, 15, 20, 23, 25, 26, 41, 42, 45, 46, 49, 62, 68, 71, 72).

[16] K. Gudka, R. N. M. Watson, J. Anderson, *et al.*, "Clean application compartmentalization with SOAAP," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 1016–1031. DOI: 10.1145/2810103.2813611. [Online]. Available: https://doi.org/10.1145/2810103.2813611 (cited on pp. 14, 19, 32, 75).

[17] N. C. Wanninger, J. J. Bowden, K. Shetty, A. Garg, and K. C. Hale, "Isolating functions at the hardware limit with virtines," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22, Rennes, France: Association for Computing Machinery, 2022, pp. 644–662, ISBN: 9781450391627. DOI: 10.1145/3492321.3519553. [Online]. Available: https://doi.org/10.1145/3492321.3519553 (cited on pp. 14, 25).

[18] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM*

SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 143–156, ISBN: 9781450375542. DOI: 10.1145/3381052.3381326. [Online]. Available: https://doi.org/10.1145/3381052.3381326 (cited on pp. 14, 15, 26).

[19] I. Agadakos, M. Egele, and W. K. Robertson, "Polytope: Practical memory access control for C++ applications," *CoRR*, vol. abs/2201.08461, 2022. arXiv: 2201.08461. [Online]. Available: https://arxiv.org/abs/2201.08461 (cited on p. 14).

[20] Y. Huang, V. Narayanan, D. Detweiler, *et al.*, "KSplit: Automating device driver isolation," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 613–631, ISBN: 978-1-939133-28-1. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/huang-yongzhe (cited on p. 14).

[21] A. Ltd., "Arm® architecture reference manual supplement morello for a-profile architecture," 2022, Accessed 14/07/2023. [Online]. Available: https://documentation-service.arm.com/static/61e577e1b691546d37bd38a0 (cited on pp. 14, 34).

[22] J. Woodruff, R. N. M. Watson, D. Chisnall, *et al.*, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, IEEE Computer Society, 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853201 (cited on pp. 14, 19, 35).

[23] R. N. M. Watson, R. M. Norton, J. Woodruff, *et al.*, "Fast protection-domain crossing in the CHERI capability-system architecture," *IEEE Micro*, vol. 36, no. 5, pp. 38–49, 2016. DOI: 10.1109/MM.2016.84. [Online]. Available: https://doi.org/10.1109/MM.2016.84 (cited on p. 14).

[24] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 721–732, ISBN: 9781450324779. DOI: 10.1145/2508859.2516713. [Online]. Available: https://doi.org/10.1145/2508859.2516713 (cited on pp. 14, 19, 32).

[25] T. Jim, J. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," English, in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02, USENIX Association, 2002, pp. 275–288, ISBN: 1-880446-00-6 (cited on pp. 14, 19, 32).

[26] U. Dhawan, A. Kwon, E. Kadric, *et al.*, "Hardware support for safety interlocks and introspection," in *Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2012, Lyon, France, September 10-14, 2012*, IEEE Computer Society, 2012, pp. 1–8. DOI: `10.1109/SASOW.2012.11`. [Online]. Available: `https://doi.org/10.1109/SASOW.2012.11` (cited on pp. 14, 32).

[27] Defence Advanced Research Projects Agency, *Broad agency announcement compartmentalization and privilege management (cpm)*, (Accessed: 14/07/2023). [Online]. Available: `https://sam.gov/opp/d624b5ffc9d24e38b966a714cbbb4f7d/view` (cited on pp. 14, 20, 44, 45, 73, 77, 79).

[28] H. Lefeuvre, V. Badoiu, Y. Chen, F. Huici, N. Dautenhahn, and P. Olivier, "Assessing the impact of interface vulnerabilities in compartmentalized software," in *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, The Internet Society, 2023. [Online]. Available: `https://www.ndss-symposium.org/ndss-paper/assessing-the-impact-of-interface-vulnerabilities-in-compartmentalized-software/` (cited on pp. 14, 20, 21, 75).

[29] D. Chisnall, B. Davis, K. Gudka, *et al.*, "CHERI JNI: sinking the java security model into the C," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds., ACM, 2017, pp. 569–583. DOI: `10.1145/3037697.3037725`. [Online]. Available: `https://doi.org/10.1145/3037697.3037725` (cited on pp. 14, 41).

[30] L. G. Esswood, "CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-961, Sep. 2021. DOI: `10.48456/tr-961`. [Online]. Available: `https:`

//www.cl.cam.ac.uk/techreports/UCAM-CL-TR-961.pdf (cited on pp. 14, 19, 39).

[31] S. Amar, T. Chen, D. Chisnall, *et al.*, "Cheriot: Rethinking security for low-cost embedded systems," Microsoft, Tech. Rep. MSR-TR-2023-6, Feb. 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/ (cited on pp. 14, 39).

[32] P. Olivier, A. Barbalace, and B. Ravindran, "The case for intra-unikernel isolation," English, The 10th Workshop on Systems for Post-Moore Architectures ; Conference date: 27-04-2020 Through 27-04-2020, Mar. 2020 (cited on pp. 15, 42).

[33] *Sqlite website*, https://www.sqlite.org/index.html, Accessed 14/07/2023, 2023 (cited on p. 15).

[34] *Libsodium website*, https://doc.libsodium.org/, Accessed 14/07/2023, 2023 (cited on p. 15).

[35] Z. Durumeric, F. Li, J. Kasten, *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. [Online]. Available: https://doi.org/10.1145/2663716.2663755 (cited on p. 18).

[36] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The race to the vulnerable: Measuring the log4j shell incident," in *6th Network Traffic Measurement and Analysis Conference, TMA 2022, Enschede, The Netherlands, June 27-30, 2022*, R. Ensafi, A. Lutu, A. Sperotto, and R. van Rijswijk-Deij, Eds., IFIP, 2022. [Online]. Available: https://dl.ifip.org/db/conf/tma/tma2022/tma2022-paper40.pdf (cited on p. 18).

[37] Accessed: 14/07/2023. [Online]. Available: https://www.cve.org/About/Metrics (cited on p. 18).

[38] MSRC Team, *A proactive approach to more secure code msrc blog microsoft security response center*, https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, (Accessed: 14/07/2023), Mar. 2023 (cited on p. 19).

[39] The Chromium Projects, *Memory safety*, https://www.chromium.org/Home/chromium-security/memory-safety/, (Accessed: 14/07/2023), May 2020 (cited on p. 19).

[40] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, P. Barham and T. Roscoe, Eds., USENIX Association, 2010. [Online]. Available: https://www.usenix.org/conference/usenix-atc-10/tolerating-malicious-device-drivers-linux (cited on p. 19).

[41] N. Roessler, L. Atayde, I. Palmer, *et al.*, "Mscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21, San Sebastian, Spain: Association for Computing Machinery, 2021, pp. 296–311, ISBN: 9781450390583. DOI: 10.1145/3471621.3471839. [Online]. Available: https://doi.org/10.1145/3471621.3471839 (cited on p. 19).

[42] Rust For Linux, *Rust for linux*, 2023 (cited on pp. 19, 28).

[43] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/ (cited on p. 19).

[44] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: Language-based restriction of untrusted libraries," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21, Virtual, USA: Association for Computing Machinery, 2021, pp. 255–267, ISBN: 9781450383172. DOI: 10.1145/3445814.3446728. [Online]. Available: https://doi.org/10.1145/3445814.3446728 (cited on pp. 19, 26).

[45] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet

Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018%5C_08-3%5C_Vasilakis%5C_paper.pdf (cited on p. 19).

[46] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., USENIX Association, 2016, pp. 549–564. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp (cited on p. 19).

[47] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith, "Ghostbusting: Mitigating spectre with intraprocess memory isolation," in *Proceedings of the 7th Annual Symposium on Hot Topics in the Science of Security, HotSoS 2020, Lawrence, Kansas, USA, September 22-24, 2020*, P. Alexander, D. Davidson, and B. Choi, Eds., ACM, 2020, 10:1–10:11. DOI: 10.1145/3384217.3385627. [Online]. Available: https://doi.org/10.1145/3384217.3385627 (cited on p. 19).

[48] S. Narayan, C. Disselkoen, D. Moghimi, *et al.*, "Swivel: Hardening WebAssembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 1433–1450, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan (cited on p. 19).

[49] A. Milburn, E. van der Kouwe, and C. Giuffrida, "Mitigating information leakage vulnerabilities with type-based data isolation," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, IEEE, 2022, pp. 1049–1065. DOI: 10.1109/SP46214.2022.9833675. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833675 (cited on p. 19).

[50] A. Prout, W. Arcand, D. Bestor, *et al.*, "Measuring the impact of spectre and meltdown," in *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, IEEE, 2018, pp. 1–5. DOI: 10.1109/HPEC.2018.8547554. [Online]. Available: https://doi.org/10.1109/HPEC.2018.8547554 (cited on p. 19).

[51] N. Abu-Ghazaleh, D. Ponomarev, and D. Evtyushkin, "How the spectre and meltdown hacks really worked," *IEEE Spectrum*, vol. 56, no. 3, pp. 42–49, 2019. DOI: 10.1109/MSPEC.2019.8651934 (cited on p. 19).

[52] M. D. Hill, J. Masters, P. Ranganathan, P. Turner, and J. L. Hennessy, "On the spectre and meltdown processor security vulnerabilities," *IEEE Micro*, vol. 39, no. 2, pp. 9–19, 2019. DOI: 10.1109/MM.2019.2897677 (cited on p. 19).

[53] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, ACM, 2019, p. 60. DOI: 10.1145/3316781.3317903. [Online]. Available: https://doi.org/10.1145/3316781.3317903 (cited on p. 19).

[54] V. Narayanan, T. Huang, D. Detweiler, *et al.*, "RedLeaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 21–39, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram (cited on pp. 19, 28).

[55] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 63–67, 2014. DOI: 10.1109/MSP.2014.66 (cited on p. 19).

[56] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A Hardware-Software co-design for efficient Intra-Enclave isolation," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3129–3145, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/gu-jinyu (cited on p. 19).

[57] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA: USENIX Association, Aug. 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/privtrans-automatically-partitioning-programs-privilege (cited on pp. 19, 24, 33).

[58] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, IEEE Computer

Society, 2016, pp. 56–71. DOI: 10.1109/SP.2016.12. [Online]. Available: https://doi.org/10.1109/SP.2016.12 (cited on pp. 19, 26).

[59] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into Reduced-Privilege compartments," in *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA: USENIX Association, Apr. 2008. [Online]. Available: https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments (cited on pp. 19, 24).

[60] H. Lee, C. Song, and B. B. Kang, "Lord of the x86 rings: A portable user mode privilege separation architecture on x86," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, 2018, pp. 1441–1454. DOI: 10.1145/3243734.3243748. [Online]. Available: https://doi.org/10.1145/3243734.3243748 (cited on p. 19).

[61] D. Sehr, R. Muth, C. Biffle, *et al.*, "Adapting software fault isolation to contemporary CPU architectures," in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, USENIX Association, 2010, pp. 1–12. [Online]. Available: http://www.usenix.org/events/sec10/tech/full%5C_papers/Sehr.pdf (cited on pp. 20, 27).

[62] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, A. D. Keromytis, Ed., USENIX Association, 2006. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture (cited on pp. 20, 27).

[63] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988, ISSN: 0163-5980. DOI: 10.1145/54289.871709. [Online]. Available: https://doi.org/10.1145/54289.871709 (cited on p. 21).

[64] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, W. Schröder-Preikschat, J. Wilkes, and R. Isaacs, Eds., ACM, 2009, pp. 219–232. DOI: 10.1145/1519065.1519090.

[Online]. Available: https://doi.org/10.1145/1519065.1519090 (cited on p. 22).

[65] J. Wang, X. Xiong, and P. Liu, "Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA: USENIX Association, Jul. 2015, pp. 361–373, ISBN: 978-1-931971-225. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/wang_jun (cited on p. 24).

[66] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-Weight contexts: An OS abstraction for safety and performance," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 49–64, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton (cited on p. 24).

[67] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The ART of app compartmentalization: Compiler-based library privilege separation on stock android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, 2017, pp. 1037–1049. DOI: 10.1145/3133956.3134064. [Online]. Available: https://doi.org/10.1145/3133956.3134064 (cited on p. 24).

[68] D. Kilpatrick, "Privman: A library for partitioning applications," in *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX: USENIX Association, Jun. 2003. [Online]. Available: https://www.usenix.org/conference/2003-usenix-annual-technical-conference/privman-library-partitioning-applications (cited on p. 24).

[69] R. Strackx, P. Agten, N. Avonds, and F. Piessens, "Salus: Kernel support for secure process compartments," *EAI Endorsed Trans. Security Safety*, vol. 2, no. 3, e1, 2015. DOI: 10.4108/sesa.2.3.e1. [Online]. Available: https://doi.org/10.4108/sesa.2.3.e1 (cited on p. 24).

[70] Y. Wu, J. Sun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *2013 28th IEEE/ACM International Conference on Automated Software*

Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, E. Denney, T. Bultan, and A. Zeller, Eds., IEEE, 2013, pp. 323–333. DOI: 10.1109/ASE.2013.6693091. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693091 (cited on pp. 24, 33, 34).

[71] Arm Ltd., *Arm architecture reference manual for a-profile architecture*, Accessed: 14/07/2023. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/ (cited on pp. 24, 25, 29).

[72] Intel Corporation, *Intel® 64 and ia-32 architectures developer's manual: Vol. 3a*, Accessed: 14/07/2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (cited on pp. 24, 26).

[73] J. Ahn, S. Jin, and J. Huh, "Fast two-level address translation for virtualized systems," *IEEE Transactions on Computers*, vol. 64, no. 12, pp. 3461–3474, 2015 (cited on p. 25).

[74] Intel Corporation, *Intel® 64 and ia-32 architectures developer's manual: Vol. 3c: System programming guide, part 3*, Accessed: 14/07/2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (cited on p. 25).

[75] Advanced Micro Devices Inc., *Amd64 architecture programmer's manual volume 2: System programming*, Accessed: 14/07/2023. [Online]. Available: https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volume-2-system-programming (cited on pp. 25, 26).

[76] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1607–1619, ISBN: 9781450338325. DOI: 10.1145/2810103.2813690. [Online]. Available: https://doi.org/10.1145/2810103.2813690 (cited on p. 25).

[77] R. J. Creasy, "The origin of the vm/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981 (cited on p. 25).

[78] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," in *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*, H. Schorr, A. J. Perlis, P. Weiner, and W. D. Frazer, Eds., ACM, 1973, p. 121. DOI: 10.1145/800009.808061. [Online]. Available: https://doi.org/10.1145/800009.808061 (cited on p. 25).

[79] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974. DOI: 10.1109/MC.1974.6323581. [Online]. Available: https://doi.org/10.1109/MC.1974.6323581 (cited on p. 25).

[80] M. Peinado, Y. Chen, P. England, and J. Manferdelli, "Ngscb: A trusted open system," in *Information Security and Privacy*, H. Wang, J. Pieprzyk, and V. Varadharajan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 86–97, ISBN: 978-3-540-27800-9 (cited on p. 25).

[81] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, B. N. Bershad and J. C. Mogul, Eds., USENIX Association, 2006, pp. 279–292. [Online]. Available: http://www.usenix.org/events/osdi06/tech/ta-min.html (cited on p. 25).

[82] Y. Li, J. M. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds., USENIX Association, 2014, pp. 409–420. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li%5C_yanlin (cited on p. 25).

[83] X. Chen, T. Garfinkel, E. C. Lewis, *et al.*, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, S. J. Eggers and J. R. Larus, Eds., ACM, 2008, pp. 2–13. DOI: 10.1145/1346281.1346284. [Online]. Available: https://doi.org/10.1145/1346281.1346284 (cited on p. 25).

[84] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, D. Gregg, V. S. Adve, and B. N. Bershad, Eds., ACM, 2008, pp. 71–80. DOI: 10.1145/1346256.1346267. [Online]. Available: https://doi.org/10.1145/1346256.1346267 (cited on p. 25).

[85] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, V. Sarkar and R. Bodík, Eds., ACM, 2013, pp. 265–278. DOI: 10.1145/2451116.2451146. [Online]. Available: https://doi.org/10.1145/2451116.2451146 (cited on p. 25).

[86] J. Criswell, N. Dautenhahn, and V. S. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds., ACM, 2014, pp. 81–96. DOI: 10.1145/2541940.2541986. [Online]. Available: https://doi.org/10.1145/2541940.2541986 (cited on p. 25).

[87] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X, San Jose, California: Association for Computing Machinery, 2002, pp. 304–316, ISBN: 1581135742. DOI: 10.1145/605397.605429. [Online]. Available: https://doi.org/10.1145/605397.605429 (cited on p. 25).

[88] E. Witchel and K. Asanović, "Hardware works, software doesn't: Enforcing modularity with mondriaan memory protection," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI: USENIX Association, May 2003. [Online]. Available: https://www.usenix.org/conference/hotos-ix/hardware-works-software-doesnt-enforcing-modularity-mondriaan-memory-protection (cited on p. 25).

[89] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V, Boston, Massachusetts, USA: As-

sociation for Computing Machinery, 1992, pp. 175–186, ISBN: 0897915348. DOI: 10.1145/143365.143508. [Online]. Available: https://doi.org/10.1145/143365.143508 (cited on p. 26).

[90] L. Delshadtehrani, S. Canakci, M. Egele, and A. Joshi, "Sealpk: Sealable protection keys for RISC-V," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, IEEE, 2021, pp. 1278–1281. DOI: 10.23919/DATE51398.2021.9473932. [Online]. Available: https://doi.org/10.23919/DATE51398.2021.9473932 (cited on p. 26).

[91] International Business Machines, *Power isa™ version 3.0 b*, Accessed: 14/07/2023. [Online]. Available: https://openpowerfoundation.org/specifications/isa/ (cited on p. 26).

[92] Arm Ltd., *Arm architecture reference manual armv7-a and armv7-r edition*, Accessed: 14/07/2023. [Online]. Available: https://developer.arm.com/documentation/ddi0406/latest/ (cited on p. 26).

[93] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "Libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 241–254, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon (cited on pp. 26, 34).

[94] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, A. P. Black and B. Liskov, Eds., ACM, 1993, pp. 203–216. DOI: 10.1145/168619.168635. [Online]. Available: https://doi.org/10.1145/168619.168635 (cited on p. 27).

[95] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke, "Compiling sandboxes: Formally verified software fault isolation," in *Programming Languages and Systems*, L. Caires, Ed., Cham: Springer International Publishing, 2019, pp. 499–524, ISBN: 978-3-030-17184-1 (cited on p. 27).

[96] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM Conference on Computer and Communications Secu-*

rity, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011, Y. Chen, G. Danezis, and V. Shmatikov, Eds., ACM, 2011, pp. 29–40. DOI: 10.1145/2046707.2046713. [Online]. Available: https://doi.org/10.1145/2046707.2046713 (cited on p. 27).

[97] L. Zhao, G. Li, B. D. Sutter, and J. Regehr, "Armor: Fully verified software fault isolation," in *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds., ACM, 2011, pp. 289–298. DOI: 10.1145/2038642.2038687. [Online]. Available: https://doi.org/10.1145/2038642.2038687 (cited on p. 27).

[98] B. Yee, D. Sehr, G. Dardyk, *et al.*, "Native client: A sandbox for portable, untrusted x86 native code," in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, IEEE Computer Society, 2009, pp. 79–93. DOI: 10.1109/SP.2009.25. [Online]. Available: https://doi.org/10.1109/SP.2009.25 (cited on p. 27).

[99] R. Sinha, M. Costa, A. Lal, *et al.*, "A design and verification methodology for secure isolated regions," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 665–681, ISBN: 9781450342612. DOI: 10.1145/2908080.2908113. [Online]. Available: https://doi.org/10.1145/2908080.2908113 (cited on p. 27).

[100] G. Morrisett, G. Tan, J. Tassarotti, J. Tristan, and E. Gan, "Rocksalt: Better, faster, stronger SFI for the x86," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds., ACM, 2012, pp. 395–404. DOI: 10.1145/2254064.2254111. [Online]. Available: https://doi.org/10.1145/2254064.2254111 (cited on p. 27).

[101] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Symposium on Operating System Design and Implementation (OSDI)*, Nov. 2006. [Online]. Available: https://www.microsoft.com/en-us/research/publication/xfi-software-guards-for-system-address-spaces/ (cited on p. 27).

[102] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with API integrity and multi-principal modules," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds., ACM, 2011, pp. 115–128. DOI: 10.1145/2043556.2043568. [Online]. Available: https://doi.org/10.1145/2043556.2043568 (cited on pp. 27, 28).

[103] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, pp. 193–204, ISBN: 9781450349444. DOI: 10.1145/3052973.3052983. [Online]. Available: https://doi.org/10.1145/3052973.3052983 (cited on p. 27).

[104] Y. Shen, H. Tian, Y. Chen, *et al.*, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 955–970, ISBN: 9781450371025. DOI: 10.1145/3373376.3378469. [Online]. Available: https://doi.org/10.1145/3373376.3378469 (cited on pp. 27, 29).

[105] "WebAssembly Core Specification," W3C, version 1.0, Dec. 5, 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/ (cited on p. 27).

[106] "WebAssembly Core Specification," W3C, version 2.0, Apr. 19, 2022, https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. [Online]. Available: https://www.w3.org/TR/wasm-core-2/ (cited on p. 27).

[107] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20, Delft, Netherlands: Association for Computing Machinery, 2020, ISBN: 9781450381536. DOI: 10.1145/3423211.3425680. [Online]. Available: https://doi.org/10.1145/3423211.3425680 (cited on p. 27).

[108] A. Zakai, *Wasmboxc: Simple, easy, and fast vm-less sandboxing*, 2020 (cited on p. 27).

[109] S. Narayan, T. Garfinkel, M. Taram, *et al.*, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 266–281, ISBN: 9781450399180. DOI: 10.1145/3582016.3582023. [Online]. Available: https://doi.org/10.1145/3582016.3582023 (cited on p. 27).

[110] S. Narayan, T. Garfinkel, E. Johnson, *et al.*, "Segue & colorguard: Optimizing sfi performance and scalability on modern x86," in *The 17th Workshop on Programming Languages and Analysis for Security*, 2022 (cited on p. 27).

[111] M. Kolosick, S. Narayan, E. Johnson, *et al.*, "Isolation without taxation: Near-zero-cost transitions for webassembly and sfi," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022. DOI: 10.1145/3498688. [Online]. Available: https://doi.org/10.1145/3498688 (cited on p. 27).

[112] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language.* Addison Wesley Professional, 2005 (cited on p. 28).

[113] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language: Everything You Need to Know.* " O'Reilly Media, Inc.", 2008 (cited on p. 28).

[114] G. Back and W. C. Hsieh, "The kaffeos java runtime system," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 583–630, Jul. 2005, ISSN: 0164-0925. DOI: 10.1145/1075382.1075383. [Online]. Available: https://doi.org/10.1145/1075382.1075383 (cited on p. 28).

[115] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder, "The JX operating system," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, C. S. Ellis, Ed., USENIX, 2002, pp. 45–58. [Online]. Available: http://www.usenix.org/publications/library/proceedings/usenix02/golm.html (cited on p. 28).

[116] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower, *J-Kernel: A Capability-Based Operating System for Java*, J. Vitek and C. D. Jensen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 369–393, ISBN: 978-3-540-48749-4. DOI: 10.1007/3-540-48749-2_17. [Online]. Available: https://doi.org/10.1007/3-540-48749-2_17 (cited on p. 28).

[117] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007, ISSN: 0163-5980. DOI: 10.1145/1243418.1243424. [Online]. Available: https://doi.org/10.1145/1243418.1243424 (cited on p. 28).

[118] B. N. Bershad, C. Chambers, S. Eggers, *et al.*, "Spin: An extensible microkernel for application-specific operating system services," ser. EW 6, Wadern, Germany: Association for Computing Machinery, 1994, pp. 68–71, ISBN: 9781450373388. DOI: 10.1145/504390.504408. [Online]. Available: https://doi.org/10.1145/504390.504408 (cited on p. 28).

[119] A. Madhavapeddy and D. J. Scott, "Unikernels: The rise of the virtual library operating system," *Communications of the ACM*, vol. 57, no. 1, pp. 61–69, 2014 (cited on pp. 28, 45).

[120] X. Leroy, D. Doligez, A. Frisch, *et al.*, "The ocaml system release 5.0: Documentation and user's manual," Inria, Tech. Rep., 2022 (cited on p. 28).

[121] C. Nichols and S. Klabnik, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019 (cited on p. 28).

[122] A. Agache, M. Brooker, A. Iordache, *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache (cited on p. 28).

[123] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, "Splinter: Bare-Metal extensions for Multi-Tenant Low-Latency storage," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 627–643, ISBN: 978-1-939133-08-3. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/kulkarni (cited on p. 28).

[124] A. Levy, M. Andersen, B. Campbell, *et al.*, "Ownership is theft: Experiences building an embedded os in rust," English (US), in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015*, ser. Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, 8th Workshop on Programming Languages and Operating Systems, PLOS 2015 ; Conference date:

04-10-2015, Association for Computing Machinery, Inc, Oct. 2015, pp. 21–26. DOI: 10.1145/2818302.2818306 (cited on p. 28).

[125] A. Light, "Reenix: Implementing a unix-like operating system in rust," *Undergraduate Honors Theses, Brown University*, 2015 (cited on p. 28).

[126] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the v out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 203–216, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda (cited on p. 28).

[127] Android Open Source Project, *Android rust introduction*, 2023 (cited on p. 28).

[128] Malwarebytes, *Windows 11 is showing its first signs of rust*, 2023 (cited on p. 28).

[129] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, "The mungi single-address-space operating system," *Software: Practice and Experience*, vol. 28, no. 9, pp. 901–928, 1998 (cited on p. 28).

[130] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Trans. Comput. Syst.*, vol. 12, no. 4, pp. 271–307, Nov. 1994, ISSN: 0734-2071. DOI: 10.1145/195792.195795. [Online]. Available: https://doi.org/10.1145/195792.195795 (cited on p. 28).

[131] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC: USENIX Association, Aug. 2010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity10/capsicum-practical-capabilities-unix (cited on pp. 28, 34).

[132] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard™: Protecting pointers from buffer overflow vulnerabilities," in *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.: USENIX Association, Aug. 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/pointguard%7B%5Ctexttrademark%7D-protecting-pointers-buffer-overflow (cited on p. 29).

[133] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 941–951. DOI: 10.1145/2810103.2813676. [Online]. Available: https://doi.org/10.1145/2810103.2813676 (cited on p. 29).

[134] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, 2017, pp. 437–452. DOI: 10.1145/3064176.3064217. [Online]. Available: https://doi.org/10.1145/3064176.3064217 (cited on pp. 29, 30).

[135] M. Gallagher, L. Biernacki, S. Chen, *et al.*, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 469–484, ISBN: 9781450362405. DOI: 10.1145/3297858.3304037. [Online]. Available: https://doi.org/10.1145/3297858.3304037 (cited on p. 29).

[136] A. Harris, T. Verma, S. Wei, *et al.*, "Morpheus II: A RISC-V security extension for protecting vulnerable software and hardware," in *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, IEEE, 2021, pp. 226–238. DOI: 10.1109/HOST49136.2021.9702275. [Online]. Available: https://doi.org/10.1109/HOST49136.2021.9702275 (cited on p. 29).

[137] OMTP Ltd., *Advanced trusted environment: Omtp tr1*, Accessed: 14/07/2023. [Online]. Available: http://www.omtp.org/ (cited on p. 29).

[138] Intel Corporation, *Intel® 64 and ia-32 architectures software developer's manual*, Accessed: 14/07/2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (cited on p. 29).

[139] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in

EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020, A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, Eds., ACM, 2020, 38:1–38:16. DOI: 10.1145/3342195.3387532. [Online]. Available: https://doi.org/10.1145/3342195.3387532 (cited on p. 29).

[140] Hex Five Security Inc., *Multizone security reference manual*, Accessed: 14/07/2023. [Online]. Available: https://github.com/hex-five/multizone-sdk/blob/master/manual.pdf (cited on p. 29).

[141] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., USENIX Association, 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan (cited on p. 29).

[142] T. V. Strydonck, J. Noorman, J. Jackson, *et al.*, "Cheri-tree: Flexible enclaves on capability machines," in *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*, IEEE, 2023, pp. 1143–1159. DOI: 10.1109/EuroSP57164.2023.00070. [Online]. Available: https://doi.org/10.1109/EuroSP57164.2023.00070 (cited on p. 29).

[143] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann (cited on p. 29).

[144] C.-c. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 645–658, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai (cited on p. 29).

[145] S. Arnautov, B. Trach, F. Gregor, *et al.*, "SCONE: Secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Associa-

tion, Nov. 2016, pp. 689–703, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov (cited on p. 29).

[146] D. Kuvaiskii, O. Oleksenko, S. Arnautov, *et al.*, "Sgxbounds: Memory safety for shielded execution," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 205–221, ISBN: 9781450349383. DOI: 10.1145/3064176.3064192. [Online]. Available: https://doi.org/10.1145/3064176.3064192 (cited on p. 29).

[147] J. Lind, C. Priebe, D. Muthukumaran, *et al.*, "Glamdring: Automatic application partitioning for intel SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 285–298, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind (cited on p. 29).

[148] S. Brenner, C. Wulf, D. Goltzsche, *et al.*, "Securekeeper: Confidential zookeeper using intel sgx," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, Trento, Italy: Association for Computing Machinery, 2016, ISBN: 9781450343008. DOI: 10.1145/2988336.2988350. [Online]. Available: https://doi.org/10.1145/2988336.2988350 (cited on p. 29).

[149] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with SGX," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, IEEE, 2020, pp. 776–789. DOI: 10.1109/ISCA45697.2020.00069. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00069 (cited on p. 29).

[150] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: efficient multi-client isolation under adversarial programs," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/chancel-efficient-multi-client-isolation-under-adversarial-programs/ (cited on p. 29).

[151] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "Twinvisor: Hardware-isolated confidential virtual machines for ARM," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds., ACM, 2021, pp. 638–654. DOI: 10.1145/3477132.3483554. [Online]. Available: https://doi.org/10.1145/3477132.3483554 (cited on p. 30).

[152] Intel Corporation, *Intel® trust domain extensions*, Accessed: 14/07/2023, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html (cited on p. 30).

[153] D. Kaplan, J. Powell, and T. Woller, "Amd sev-snp: Strengthening vm isolationwith integrity protection and more," Technical Report. Advanced Micro Devices Inc., Tech. Rep., 2020 (cited on p. 30).

[154] ARM Ltd., *Arm cca security model 1.0*, Accessed: 14/07/2023, 2021. [Online]. Available: https://developer.arm.com/documentation/DEN0096/latest/ (cited on p. 30).

[155] G. D. H. Hunt, R. Pai, M. V. Le, *et al.*, "Confidential computing for openpower," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21, Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 294–310, ISBN: 9781450383349. DOI: 10.1145/3447786.3456243. [Online]. Available: https://doi.org/10.1145/3447786.3456243 (cited on p. 30).

[156] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, "Sevurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, IEEE, 2020, pp. 1483–1496. DOI: 10.1109/SP40000.2020.00080. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00080 (cited on p. 30).

[157] K. Serebryany, "ARM memory tagging extension and how it improves C/C++ memory safety," *login Usenix Mag.*, vol. 44, no. 2, 2019. [Online]. Available: https://www.usenix.org/publications/login/summer2019/serebryany (cited on p. 30).

[158] K. Aingaran, S. Jairath, G. Konstadinidis, *et al.*, "M7: Oracle's next-generation sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar.

2015, ISSN: 0272-1732. DOI: 10.1109/MM.2015.35. [Online]. Available: https://doi.org/10.1109/MM.2015.35 (cited on p. 30).

[159] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds., USENIX Association, 2008, pp. 225–240. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full%5C_papers/zeldovich/zeldovich.pdf (cited on p. 30).

[160] M. Unterguggenberger, D. Schrammel, P. Nasahl, R. Schilling, L. Lamster, and S. Mangard, "Multi-tag: A hardware-software co-design for memory safety based on multi-granular memory tagging," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, J. K. Liu, Y. Xiang, S. Nepal, and G. Tsudik, Eds., ACM, 2023, pp. 177–189. DOI: 10.1145/3579856.3590331. [Online]. Available: https://doi.org/10.1145/3579856.3590331 (cited on p. 30).

[161] J. A. González, "Taxi: Defeating code reuse attacks with tagged memory," Ph.D. dissertation, Massachusetts Institute of Technology, 2015 (cited on p. 30).

[162] Clang team, *Hardware-assisted addresssanitizer design documentation — clang 11 documentation*, Accessed: 14/07/2023. [Online]. Available: https://releases.llvm.org/11.0.0/tools/clang/docs/%20HardwareAssistedAddressSan html (cited on p. 30).

[163] D. P. McKee, Y. Giannaris, C. Ortega, *et al.*, "Preventing kernel hacks with hakcs," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*, The Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-257/ (cited on p. 30).

[164] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the intel MPX system stack," in *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2018, Irvine, CA, USA, June 18-22, 2018*, K. Psounis, A. Akella, and A. Wierman, Eds., ACM,

2018, pp. 111–112. DOI: 10.1145/3219617.3219662. [Online]. Available: https://doi.org/10.1145/3219617.3219662 (cited on p. 30).

[165] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, IEEE, 2020, pp. 1153–1166. DOI: 10.1109/MICRO50266.2020.00095. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00095 (cited on p. 30).

[166] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, "Heapcheck: Low-cost hardware support for memory safety," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–24, 2022 (cited on p. 30).

[167] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, Portland, Oregon: IEEE Computer Society, 2012, pp. 189–200, ISBN: 9781450316422 (cited on p. 30).

[168] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, D. R. Kaeli and T. Moseley, Eds., ACM, 2014, p. 175. [Online]. Available: https://dl.acm.org/citation.cfm?id=2544147 (cited on p. 30).

[169] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, S. J. Eggers and J. R. Larus, Eds., ACM, 2008, pp. 103–114. DOI: 10.1145/1346281.1346295. [Online]. Available: https://doi.org/10.1145/1346281.1346295 (cited on p. 31).

[170] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, IEEE, 2020, pp. 762–775. DOI: 10.1109/

ISCA45697.2020.00068. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00068 (cited on pp. 31, 32).

[171] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966, ISSN: 0001-0782. DOI: 10.1145/365230.365252. [Online]. Available: https://doi.org/10.1145/365230.365252 (cited on p. 31).

[172] A. J. Mayer, "The architecture of the burroughs b5000: 20 years later and still ahead of the times?" *ACM SIGARCH Computer Architecture News*, vol. 10, no. 4, pp. 3–10, 1982 (cited on p. 31).

[173] D. England, "Capability concept mechanism and structure in system 250," in *Proceedings of the International Workshop on Protection in Operating Systems*, 1974, pp. 63–82 (cited on p. 31).

[174] R. D. H. Walker, "The structure of a well-protected computer," Ph.D. dissertation, University of Cambridge, UK, 1973 (cited on p. 31).

[175] M. V. Wilkes and R. M. Needham, "The cambridge cap computer and its operating system," 1979 (cited on p. 31).

[176] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, F. Baskett and D. W. Clark, Eds., ACM Press, 1994, pp. 319–327. DOI: 10.1145/195473.195579. [Online]. Available: https://doi.org/10.1145/195473.195579 (cited on p. 31).

[177] K. Ghose and P. Vasek, "A fast capability extension to a RISC architecture," in *22rd EUROMICRO Conference '96, Beyond 2000: Hardware and Software Design Strategies, September 2-5, 1996, Prague, Czech Republic*, IEEE Computer Society, 1996, p. 606. DOI: 10.1109/EURMIC.1996.546488. [Online]. Available: https://doi.org/10.1109/EURMIC.1996.546488 (cited on p. 31).

[178] P. Vasek and K. Ghose, "A comparison of two context allocation approaches for fast protected calls," in *Proceedings of the Fourth International on High-Performance Computing, HiPC 1997, Bangalore, India, 18-21 December, 1997*, IEEE Computer Society, 1997, pp. 16–21. DOI: 10.1109/HIPC.1997.634463. [Online]. Available: https://doi.org/10.1109/HIPC.1997.634463 (cited on p. 31).

[179] R. N. Watson, P. G. Neumann, J. Woodruff, *et al.*, "Capability hardware enhanced risc instructions: Cheri instruction-set architecture," University of Cambridge, Computer Laboratory, Tech. Rep., 2020 (cited on pp. 32–35).

[180] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "Codoms: Protecting software with code-centric memory domains," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, IEEE Computer Society, 2014, pp. 469–480. DOI: 10.1109/ISCA.2014.6853202. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853202 (cited on p. 32).

[181] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API usages through semantic Cross-Checking," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 363–378, ISBN: 978-1-931971-32-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun (cited on pp. 33, 76).

[182] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure inter-process communication for microkernels," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, Dresden, Germany: Association for Computing Machinery, 2019, ISBN: 9781450362818. DOI: 10.1145/3302424.3303946. [Online]. Available: https://doi.org/10.1145/3302424.3303946 (cited on p. 33).

[183] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA: USENIX Association, Jun. 2001. [Online]. Available: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/integrating-flexible-support-security-policies (cited on p. 34).

[184] R. M. Norton, "Hardware support for compartmentalisation," Ph.D. dissertation, University of Cambridge, Computer Laboratory, 2016 (cited on p. 34).

[185] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and unlimited virtual domains on multiple architectures," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASP-

LOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 905–919, ISBN: 9781450399166. DOI: 10.1145/3575693.3575735. [Online]. Available: https://doi.org/10.1145/3575693.3575735 (cited on p. 34).

[186] ARM Ltd., *Arm Morello System Development Platform (SDP) Technical Reference Manual*, Accessed: 14/07/2023, 2022. [Online]. Available: %5Curl%7Bhttps://developer.arm.com/documentation/102278/0000%7D (cited on pp. 34, 67).

[187] A. Pellegrini, N. Stephens, M. Bruce, *et al.*, "The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020 (cited on p. 34).

[188] G. Kane and J. Heinrich, *MIPS RISC architectures*. Prentice-Hall, Inc., 1992 (cited on p. 35).

[189] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "ISA semantics for armv8-a, risc-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 71:1–71:31, 2019. DOI: 10.1145/3290384. [Online]. Available: https://doi.org/10.1145/3290384 (cited on p. 35).

[190] J. Woodruff, A. Joannou, H. Xia, *et al.*, "Cheri concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, 2019 (cited on p. 35).

[191] H. Xia, J. Woodruff, H. Barral, *et al.*, "Cherirtos: A capability model for embedded devices," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, 2018, pp. 92–99 (cited on p. 38).

[192] B. Davis, R. N. M. Watson, A. Richardson, *et al.*, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, 2019, pp. 379–393. DOI: 10.1145/3297858.3304042. [Online]. Available: https://doi.org/10.1145/3297858.3304042 (cited on p. 40).

[193] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, *et al.*, "Unikraft: Fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21, Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 376–394, ISBN:

9781450383349. DOI: 10.1145/3447786.3456248. [Online]. Available: https://doi.org/10.1145/3447786.3456248 (cited on pp. 41, 42, 46).

[194] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "EbbRT: A framework for building Per-Application library operating systems," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 671–688, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg (cited on p. 41).

[195] A. Kivity, D. Laor, G. Costa, *et al.*, "OSv—Optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72, ISBN: 978-1-931971-10-2. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity (cited on p. 41).

[196] J. Martins, M. Ahmed, C. Raiciu, *et al.*, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, R. Mahajan and I. Stoica, Eds., USENIX Association, 2014, pp. 459–473. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins (cited on p. 41).

[197] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 339–354 (cited on p. 41).

[198] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 59–73, ISBN: 9781450360203. DOI: 10.1145/3313808.3313817. [Online]. Available: https://doi.org/10.1145/3313808.3313817 (cited on p. 41).

[199] A. Madhavapeddy, R. Mortier, C. Rotsos, *et al.*, "Unikernels: Library operating systems for the cloud," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, V. Sarkar and R. Bodík, Eds., ACM, 2013, pp. 461–472. DOI: 10.1145/2451116.2451167. [Online]. Available: https://doi.org/10.1145/2451116.2451167 (cited on p. 41).

[200] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10.1145/3342195.3387526. [Online]. Available: https://doi.org/10.1145/3342195.3387526 (cited on p. 41).

[201] O. Purdila, L. A. Grijincu, and N. Tapus, "Lkl: The linux kernel library," in *9th RoEduNet IEEE International Conference*, 2010, pp. 328–333 (cited on p. 41).

[202] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 291–304, ISBN: 9781450302661. DOI: 10.1145/1950365.1950399. [Online]. Available: https://doi.org/10.1145/1950365.1950399 (cited on p. 41).

[203] S. Kuenzer, A. Ivanov, F. Manco, *et al.*, "Unikernels everywhere: The case for elastic cdns," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17, Xi'an, China: Association for Computing Machinery, 2017, pp. 15–29, ISBN: 9781450349482. DOI: 10.1145/3050748.3050757. [Online]. Available: https://doi.org/10.1145/3050748.3050757 (cited on p. 41).

[204] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10.1145/3342195.3392698. [Online]. Available: https://doi.org/10.1145/3342195.3392698 (cited on p. 41).

[205] H. Lefeuvre, V.-A. Bădoiu, Ş. Teodorescu, *et al.*, "Flexos: Making os isolation flexible," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 79–87, ISBN: 9781450384384. DOI: 10.1145/3458336.3465292. [Online]. Available: https://doi.org/10.1145/3458336.3465292 (cited on p. 42).

[206] R. N. Watson, A. Richardson, B. Davis, *et al.*, "Cheri c/c++ programming guide," University of Cambridge, Computer Laboratory, Tech. Rep., 2020 (cited on p. 48).

[207] C. A. Thekkath and H. M. Levy, "Hardware and software support for efficient exception handling," in *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, F. Baskett and D. W. Clark, Eds., ACM Press, 1994, pp. 110–119. DOI: 10.1145/195473.195515. [Online]. Available: https://doi.org/10.1145/195473.195515 (cited on p. 52).

[208] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, R. H. Arpaci-Dusseau and B. Chen, Eds., USENIX Association, 2010, pp. 33–46. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full%5C_papers/Soares.pdf (cited on p. 74).

[209] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, "Arbitrar: User-guided api misuse detection," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1400–1415. DOI: 10.1109/SP40001.2021.00090 (cited on p. 76).