Software Compartmentalization Trade-Offs with Hardware Capabilities

John Alistair Kressel, Hugo Lefeuvre, Pierre Olivier
The University of Manchester
Manchester, UK

Abstract

Compartmentalization is a form of defensive software design in which an application is broken down into isolated but communicating components. Retrofitting compartmentalization into existing applications is often thought to be expensive from the engineering effort and performance overhead points of view. Still, recent years have seen proposals of compartmentalization methods with promises of low engineering efforts and reduced performance impact. ARM Morello combines a modern ARM processor with an implementation of Capability Hardware Enhanced RISC Instructions (CHERI) aiming to provide efficient and secure compartmentalization. Past works exploring CHERI-based compartmentalization were restricted to emulated/FPGA prototypes.

In this paper, we explore possible compartmentalization schemes with CHERI on the Morello chip. We propose two approaches representing different trade-offs in terms of engineering effort, security, scalability, and performance impact. We describe and implement these approaches on a prototype OS running bare metal on the Morello chip, compartmentalize two popular applications, and investigate the performance overheads. Furthermore, we show that compartmentalization can be achieved with an engineering cost that can be quite low if one is willing to trade off on scalability and security, and that performance overheads are similar to other intra-address space isolation mechanisms.

CCS Concepts: • **Security and privacy → Security in hardware**; **Operating systems security**; **Software and application security**.

Keywords: Compartmentalization, Hardware Capabilities

1 Introduction

Software compartmentalization is one of the ways to enforce the principle of least privilege [31]. Compartmentalization enforces isolation between components of a software system, granting compartments only the minimal privileges they need to function. If a component of a compartmentalized system is subverted, the damage the attacker can do is limited to the privileges granted to the compromised compartment [17, 30]. Contrary to many other protection techniques, compartmentalization allows defending against yet unknown/future vulnerabilities in existing code bases [41]. Many approaches have been proposed in recent years, utilizing different hardware and software isolation mechanisms to compartmentalize libraries [5, 6, 15, 22, 25, 28, 32–34, 37, 44] as well as smaller pieces of code such as functions [3, 14, 36, 38].

Morello [26] is an extension to the ARMv8-A architecture implementing the Capability Hardware Enhanced RISC Instructions (CHERI), designed specifically to enable high-performance and scalable compartmentalization [40, 41, 43]. This is achieved by enforcing compartment bounds on most memory loads and stores in hardware, and letting communicating compartments securely lend memory to each other using so-called hardware capabilities, a mechanism similar to fat pointers [20] implemented in hardware to restrict accesses to shared memory at a fine (byte-level) granularity.

When retrofitting compartmentalization to existing code bases, a key challenge is keeping refactoring costs low [4]. This is crucial not only for reducing the cost of deployment, but also to reduce the number of errors made during the compartmentalization, which can undermine its efficiency or security guarantees [21]. Work exploring compartmentalization with CHERI is so far limited to solutions with relatively high porting costs [11, 32, 41], that require a non-negligible reworking to the code corresponding to inter-compartment communications. 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 an ASIC processor. In that context, the recent availability of Morello raises the research questions we tackle in this paper:

1. Which compartment models are possible using Morello, using what programming abstractions, at which refactoring costs?
2. How does Morello's compartmentalization performance and security guarantees compare to other intra-address space compartmentalization mechanisms (e.g., MPK)?

For this purpose, we adapt an existing compartmentalization-oriented library OS (libOS), FlexOS [22], to Morello, and extend it by developing two compartmentalization programming abstractions relying on hardware capabilities, each representing a particular trade-off in terms of porting costs, security guarantees, and scalability to multiple compartments. The first is based on manual sandboxing as advocated by CHERI's designers [41], with every shared buffer protected by a capability. Further, we propose a second approach relying on a single region of shared data between two mutually distrusting compartments. These abstractions are used to compartmentalize popular open source software, SQLite [2] and LibSodium [1], at different isolation granularities: functions and libraries. We evaluate the porting costs, degree of security of these solutions, and further evaluate their performance when executing on the Morello chip, comparing these results to that of another intra-address space isolation mechanism: Intel MPK. We show that manual porting as advocated by CHERI's designers offers good performance, a good level of security, and scales well to high numbers of compartments. However, it can require a significant engineering effort when applied to large compartments. The second approach trades off security guarantees and scalability to more than 2 compartments, to achieve low porting costs, requiring only annotations indicating shared data at declaration time in the code.

2 CHERI Hardware Capabilities

A hardware *capability* [43] is an architectural data type used to represent a contiguous region of virtual memory with byte-level granularity. CHERI hardware capabilities define a base address, bounds and permissions information. The capability can be dereferenced to access the memory it refers to, with the hardware performing bounds and permissions checks. Capabilities are made unforgeable by a validity tag, stored separately, and the restriction that capabilities can only be used/manipulated via capability-aware instructions.

When using Morello for compartmentalization (commonly referred to as *hybrid* mode), all compartments share a single address space, and the vast majority of the program's machine code is unchanged, consisting of traditional ARMv8 instructions. Every memory access made by a core is constrained by two global capabilities that delimit the memory regions the currently executing compartment can access: the *Program Counter Capability* (PCC) and the *Default Data Capability* (DDC). There is one PCC and one DDC register per core holding these, restricting the ARMv8 code's ability to perform instruction (PCC) and data (DDC) memory accesses within the relevant bounds. Additional capabilities can be used for sharing data between compartments. That way, the caller is lending access to the smallest region of memory (the data structure) needed by the callee. This is secure due to the fine-grained bounds enforcement and efficient as no data copy happens. Capabilities are also used to control exceptionless security domains (compartment) switches, realized by a privileged security monitor (referred to as the *switcher* in this paper). This is achieved through a special type of capability referred to as *sealed*, which is immutable and non-dereferenceable, and can only be unsealed via a jump to a pre-determined instruction in the switcher.

3 Design

We propose two design approaches suitable for compartments on Morello, guided by four main considerations: the engineering effort to retrofit compartmentalization in existing software, compartmentalization's performance overhead, the security of the given approach, and how it scales to many compartments.

Engineering effort represents the effort to retrofit isolation into legacy software, or to write new compartmentalized software, using a given abstraction. It consists of marking compartment boundaries and shared data [22], e.g. with annotations, but also sometimes redesigning part of the software with security in mind [21]. If high, it can be a significant barrier to the adoption of compartmentalization, as it increases costs and development complexity [22]. *Performance overheads* are another factor hindering the popularity of compartmentalization [29, 37] and should be minimized. For example, the recent DARPA Compartmentalization and Privilege Management call [4] specifies this overhead to be <5% for application-level compartmentalization with function granularity compartments. Higher overhead is allowed for proportionally higher security gains. *Security* is a spectrum of guarantees over an uncompartmentalized system. The precise security requirements must be judged against the other requirements to strike an acceptable balance. As a minimum, solutions must enforce strong isolation between compartments and provide access to only a subset of data which has been selectively shared for communication. Finally, the *scalability* of a solution denotes its capacity to efficiently scale to many compartments.

3.1 Design Overview

In line with Morello/CHERI single address space compartmentalization model [41], and with many existing works [15, 22, 32, 33, 36, 37], we assume a libOS-based environment in which two or more user space and/or kernel components share a single address space and are isolated from each other. Compartments are defined statically at build time: each compartment is given a pair of memory regions, each contiguous in the virtual address space, to hold its private 1) code and 2) data. Global compartment capabilities constraining each compartment's memory accesses to the corresponding pair of regions are initialized at boot time. The static memory layout of the application and the systems software's dynamic memory allocation primitives (`malloc`/`mmap`/`brk`) are designed to allocate data accordingly. All data declared in the scope of a given compartment is treated as private to that compartment, unless it is specifically annotated as shared. Shared data is managed differently in the two abstractions we propose, and this is presented in detail in the next subsection.

2

```
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 1. Example of a function annotated to use capabilities.

Gates are inserted in the code in place of function calls, where these calls now cross compartment boundaries. They invoke the switcher, which performs, on the relevant CPU core, the security domain switch by switching the stack and global compartment code/data capability registers. Finally, a trampoline is called to enable fast return to the caller compartment from capability-unaware code, with less overhead than re-invoking the switcher. The switcher is trusted and privileged, hence its code and data (including the global capabilities for all compartments) cannot be accessed by the compartments directly, instead they must use a sealed capability (§2). The switching mechanism is kept as lightweight as possible to minimize overhead while preserving strong security. We define the trusted computing base as the switcher, gates, trampoline, early boot code (including capability initialization code), memory manager, scheduler and interrupt handler.

3.2 Two Approaches To Sharing Data

Compartments cannot access memory outside the regions constrained by their *DDC/PCC*. This raises the issue of how to selectively share data between compartments, and how to do so efficiently (i.e. without data copy). We discuss two approaches to data sharing, and reason about their performance, security, and scalability properties.

3.2.1 Approach 1: Replacing Pointers with Capabilities. With CHERI, certain pointers can be transformed into fine-grained capabilities encompassing only the pointed data structure(s). This is the standard way to manage shared data as designed by the CHERI/Morello authors [41]. A compartment *C1* wishing to share a subset of its dedicated memory region with another compartment *C2* can pass such a fine-grained capability *cap* as parameter/return value of a gate. Morello is designed so that the memory access made by *C2* dereferencing *cap* will not be subject to *C2*'s global capability: *C2* will thus be able to access *C1*'s memory region, but only the few bytes represented by *cap*.

Engineering Cost. The sandboxing effort of a legacy function foo is illustrated on Listing 1. As described earlier, pointer
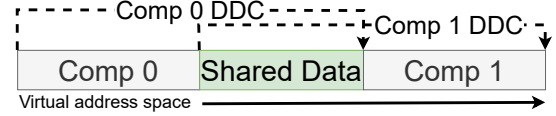

Figure 1. Compartment bounds with shared memory regions. The compartment bounds overlap to encompass shared data.

parameters (stat line 8) and any pointers created out of a capability (str line 9, 11, 12) must be transformed into capabilities with annotations. Capabilities flowing out of the compartment must be changed back into pointers with a cast (line 10). Capability monotonicity must also be respected, i.e. 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 [42]. In the general case, we hint that such manual porting may only be amenable to small-scale scenarios (e.g. sandboxing one or a few functions), because the engineering cost of rewriting pointers into capabilities becomes too high as compartments' sizes increase.

Trust Model. Engineering costs can be kept low with this approach in scenarios with 1) small compartment sizes, which limits the amount of code rewriting within the compartments; and 2) no capability flowing outside the compartment, to avoid costly rewriting of the data flow in the rest of the application. This fits very well function sandboxing scenarios. The isolated function represents a distrusted compartment, and is isolated from the rest of the system. Pointer arguments entering the compartment are replaced with capabilities. The rest of the system is trusted and can access any memory within the sandbox compartment.

Performance, Security, and Scalability. Data is shared through capabilities, leading to a low performance impact (no copy or marshaling). In terms of security, the sandbox is isolated by the *PCC/DDC*, constraining all non-capability operations made by the sandbox, and shared data is tightly bounded by argument capabilities, resulting in strong isolation. Regarding scalability, this approach scales to an unlimited number of compartments (e.g. function sandboxes) with a constant porting effort (porting complexity does not grow with the number of compartments).

3.2.2 Approach 2: Overlapping Shared Region. This approach drops the fine-grained capabilities to rely on a single region of shared data. The *DDC* bounds of communicating compartments are extended to cover this region, so both can access shared data, as illustrated in Figure 1. Since capability bounds must cover contiguous memory, the shared data region is located between two compartments in memory. The linker and dynamic memory allocation primitives ensure that shared data is correctly placed in the relevant memory.

Engineering Cost. With this approach shared data needs to be marked as such with annotations in the source code.

```
1  void foo() { // Original    6  void foo() { // Ported
2    int x;                    7    int __shared x;
3    bar(&x);                  8    __gate(bar, &x,
4  }                           9       compartment1);
5                             10  }
```

Listing 2. Example of annotations for shared regions.

The function calls at compartment boundaries are similarly annotated. This is illustrated on Listing 2, where foo and bar are placed in different compartments. Code transformations use these annotations to automatically allocate shared data in memory which is accessible from all compartments, and to instantiate gates. The engineering effort of this approach is relatively low, and significantly lower than replacing pointers with capabilities, as data must only be annotated at declaration and/or allocation sites.

Trust Model. Mutual distrust is enforced between compartments, with none able to access the others' private data.

Performance, Security, and Scalability. Unlike Approach 1, shared stack variables must be allocated on a heap in the shared region, resulting in an additional allocation cost. Techniques to address this problem like data shadow stacks [22] cannot be applied as-is due to the requirements of the *DDC*. Nevertheless, we expect performance to be comparable to the previous approach. Regarding security, isolation of memory accesses is also enforced by the compartment *PCC* and *DDC*. Data sharing is however made at a coarser granularity, with the entire shared memory region accessible to both compartments at all times. This trades off security in two ways: 1) bounds are not tight to individual objects, thus not offering CHERI's spatial safety for shared objects; 2) even assuming no revocation in Approach 1, the number of objects effectively accessible by each compartment at any execution time will remain larger, resulting in more potential for compartment interface vulnerabilities [21]. In terms of scalability, this approach only scales to a small number of compartments: indeed one can create only a single overlapping region per pair of communicating compartment, hence a scenario with e.g. 3 compartments wishing to access a shared data structure is not possible.

4 Implementation

We have selected FlexOS [22, 23], a compartmentalization-focused library operating system, to implement a prototype system. FlexOS originally supported isolation with Intel Memory Protection Keys and Extended Page Tables. The OS allows easy extension to new isolation mechanisms. Further, its design is based on the Unikraft [19] unikernel, so it inherits its high performance, small attack surface, and good compatibility with popular applications. We ported FlexOS to the Morello platform, and implemented on top of it the two compartmentalization abstractions described earlier, in a total of

about 2200 lines of code. Below we give implementation details regarding the system's initialization, the compartments' structure, and the security domain switching process.

4.1 Compartment Structure

Compartments are defined at build time in a configuration file provided to the FlexOS build tool. At link time, isolated data are placed into their respective, separate, non overlapping ELF section with the help of a custom linker script automatically generated by the toolchain. Non-isolated data are placed into a default compartment. The linker script also reserves space for dynamically-allocated data (stack and heap) in each compartment's memory.

The compartment switcher's code and data is isolated from all other compartments' code. This is done to control access to the switcher, which is a privileged entity. In addition, compartment capability pairs (one *DDC* and *PCC* pair per compartment) are stored in memory which is not accessible from any compartment but that of the switcher. This is done to avoid a compartment arbitrarily granting itself access to another compartment's memory.

4.2 Initialization

Based on the compartment boundaries defined in the linker script, compartments are initialized at boot time: we trust the boot code of FlexOS to initialize compartment capabilities correctly. Compartment's *PCC* and *DDC* bounds are initialized to cover the statically defined compartment memory region. Capability bounds can only cover a contiguous region of memory, meaning that all of the code and data of a compartment must be present in contiguous memory. Once compartment capabilities have been created, they are stored in the memory reserved for compartment capability pairs.

During boot time, a capability pair for the switcher is also initialized. This pair grants access to the switcher code, and the compartment capability pairs. To prevent unauthorized 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, a sealed capability is created for each compartment, which is unsealed using a 1pb (load pair and branch) instruction. The sealed capability is thus the only way for compartments to invoke the switcher. Each compartment is given one such sealed capability. Finally, each compartment receives a private allocator, which manages the per-compartment portion of the virtual address space previously reserved in the linker script. Using this allocator, a private stack and heap for each compartment are initialized.

At the end of the boot process, the capability pair for the default compartment is loaded and execution then enters the default compartment.

4.3 Switching Security Domains

The security domain switch process is illustrated on Figure 2: the caller compartment invokes the privileged switcher safely through a sealed capability, which switches the architectural state representing compartment permissions on the
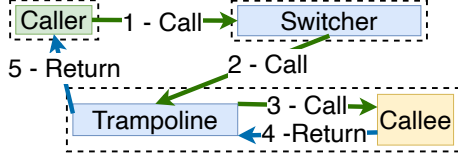
4

Figure 2. Control flow of a compartment switch (call and return paths). Dashed boxes represent protection domains. The trampoline is available in the domain of the callee compartment.

relevant CPU core. Using a trampoline, the switcher then branches to the callee compartment. On the return path, a trampoline is used to branch back to the caller.

We implement compartment switch gates as C macros. This allows the instructions invoking the switcher to be directly inlined at the call site, avoiding the need for a function call. The call gate is in the caller compartment. Unlike in other implementations of FlexOS call gates, such as MPK [22], the domain transition is not realized within the security context of the caller: this is done to prevent compartments from accessing the capability pairs of other compartments. Instead, it invokes the switcher after having loaded the parameters needed for the callee. 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. The procedure used to invoke the compartment switcher is as follows: caller-saved registers are pushed to the stack, the current stack and frame pointers 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. By invoking the switcher, 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 the only capability able to execute switcher code, which is isolated 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 are 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 (PCC, DDC) are loaded/set.
4. The stack is switched for the callee.
5. 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 caller compartment can only be performed via the capability pair stored on the caller stack, accessed via a sealed capability. This avoids the need to go through the switcher on the return path. At call time, the trampoline stores the sealed capability created by the switcher onto the callee's stack before calling the target function. Upon return to the trampoline, the sealed capability is popped, unsealed, and a return to the caller compartment is performed. The unsealed capability is used to load the caller compartment capability pair: the *PCC* is set as part of the return and the *DDC* is restored by the call gate in the caller. Upon return from the callee compartment, the gate restores the stack and any saved registers. If the function call returned a value, the gate will store the returned value in a provided variable pointer.

5 Evaluation

In this section we evaluate the impact on performance and engineering effort of our proposed approaches. We use the abstractions we developed to compartmentalize two popular applications, the SQLite [2] database management system and the libsodium [1] crypto library. For libsodium we use our first approach to data sharing, function sandboxing with fine-grained capabilities, and sandbox 5 functions manipulating external input, listed in Table 1. The library is integrated with a benchmark we derived from its test suite, running representative tests (e.g. encrypting a buffer, generating a key) 200 times in a loop. SQLite's compartmentalization uses our second approach, coarse-grain shared data regions with overlapping DDCs between communicating compartments. We create two mutually distrusting compartments: the filesystem management code, and the rest of the system. This application is benchmarked with 5000 INSERT operations on an in memory (ramfs) database. The compartmentalization scenario and the benchmark are both taken from the FlexOS paper [22] and represent a system call intensive application.

We run all experiments on our port of FlexOS, bare-metal on the Morello evaluation board [24, 39] with 16 GB of RAM and the capability-enabled SoC clocked at 2.5 GHz. For comparison, we also gather data for Linux on Morello, running a capability-unaware AArch64 Debian 11, as well as for the other isolation mechanisms supported by FlexOS (MPK, EPT) on an x86-64 Xeon Silver 4114 clocked at 2.2 Ghz with 128 GB of RAM, running Debian 11. Results are averages of 10 runs; since they show little variance we omit error bars.

5.1 Engineering Cost

Table 1 shows the porting effort associated with the compartmentalization of libsodium and SQLite.

Concerning libsodium, the engineering cost of sandboxing functions with our first approach involves rewriting the function's code to be capability-aware, something that can be a task of non-negligible complexity [42]. Still, because we deliberately selected functions with relatively small sizes (11 to 141 LoC), that effort was relatively low (1 or 2 hours per

5

Table 1. Porting effort required to compartmentalize.

| 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
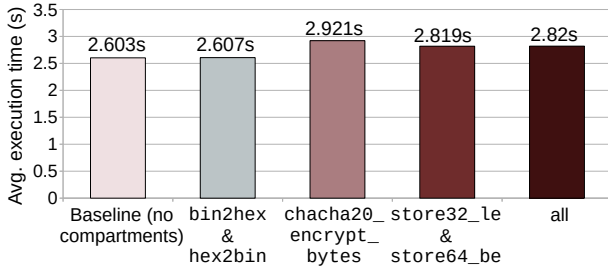
Figure 3. Overhead of various compartmentalization scenario on libsodium (X labels are sandboxed functions).

function for a programmer with good knowledge of capability programming), and mostly consisted of annotating the relevant pointers to be transformed into capabilities.

Regarding SQLite, we achieved the compartmentalization in a couple of days using the overlapping shared data region approach. The effort involved can be broken down into two tasks: 1) gate insertion and 2) shared data identification. 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 identifying shared data. It is currently with FlexOS a manual process, during which the programmer must analyze the code carefully to pinpoint what needs to be shared with annotations. Although the engineering cost for this approach seems higher than for the sandboxing method, the compartment size is also much larger for the overlapping DDC, e.g. 5.8K LoC for the filesystem compartment.

5.2 Performance

Libsodium. We analyze different configurations of the 5 `Libsodium` functions we sandboxed by replacing pointers with capabilities. The results are presented on Figure 3. The overhead is very modest, due to a relatively low amount of compartment switches; 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 compartmentalizing `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
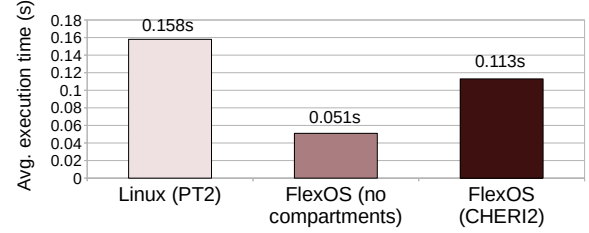
Figure 4. Execution times of different configurations of SQLite running on Morello.

the overhead is higher. Evaluating against the DARPA requirements for function granularity isolation [4], most of these results are in range of the required 5% overhead.

To further understand this behavior, we enabled hardware performance counters support in our prototype OS and gathered the data presented in Table 2. The number of instructions executed, memory accesses performed including cache accesses/misses, and branches executed, increase proportionally. The reduction in instruction cache misses for `store32_le` and `store64_be` is due to the new wrapper function being no longer inlined as the original functions were, resulting in more efficient cache utilization. We also note that the branch predictor struggles with increased use of indirect branches, a direct consequence of gates as a layer of indirection.

SQLite. Results for SQLite are presented on Figure 4. Here the overlapping shared data region approach is used for CHERI configurations. Compared to an uncompartmentalized FlexOS baseline, isolating the filesystem adds an overhead of 119.9%. This is because the isolated code lies on the hot path, meaning that isolated primitives are frequently called: we measured a high frequency of domain transitions (2.49/1k instructions). However, CHERI-based filesystem isolation still outperforms the same benchmark running on an unmodified Debian Linux installation by a factor of 1.4x. Running on Debian Linux is equivalent to a relatively costly two-compartments pagetable-based scenario (PT2), due to the page table-based user-kernel separation.

The data measured from the hardware performance counters for SQLite is presented in Table 2. Compartmentalization 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 access increase, while the number of misses for both increase by a smaller proportion. Interestingly, the branch misprediction rate rises by a far greater amount than the number of branches executed. This may be attributed to the increased number of indirect branches used as a result of the switching process, which are harder for the predictor to predict.

The relative overhead figures for SQLite are presented in Figure 5. We also include numbers for FlexOS running on x86-64 with MPK and EPT. We present results for the previously-mentioned compartmentalization scenario, 2 compartments (filesystem and rest of the system, CHERI2/PT2/EPT2), and

6

Table 2. Performance counters by for each configuration compared to the uncompartmentalized baseline.

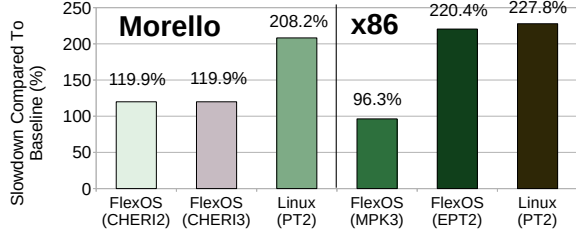| Configuration	L1I Acc	L1I Misses	L1D Acc	L1D Misses	Br Ret	Br Mispred	Mem Acc	Inst Ret
`sodium_hex2bin` & `sodium_bin2hex`	+0.15%	+3.95%	+0.09%	+25.11%	+0.29%	+10.25%	+0.13%	+1.74%
`chacha20_encrypt_bytes`	+23.58%	**+209.49%**	+16.85%	**+106.00%**	+23.33%	+76.76%	+16.69%	+4.90%
`store32_le` & `store64_be`	+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%

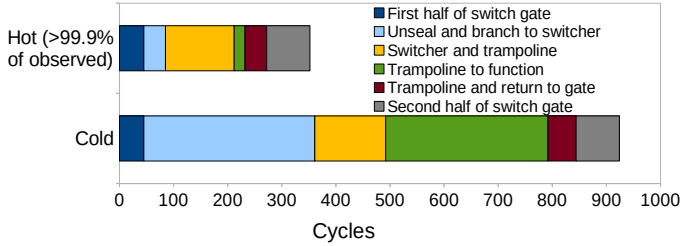**Figure 5.** Overhead relative to uncompartmentalized FlexOS for SQLite.

Figure 6. Hot and cold compartment switch latencies, broken down into component parts.

an additional scenario with 3 compartments (filesystem, time manager, rest of the system, CHERI3/MPK3). Morello's numbers are in line with the relative overheads of these other isolation mechanisms. Compared to the overhead of MPK3, CHERI is slightly more expensive. This can be attributed to the switching mechanism that with CHERI requires additional bookkeeping and jumps (e.g. to the switcher) compared to MPK.

Security Domain Switch Latency Breakdown. We observe that the majority of the overhead comes from security domain switches [35], hence we use microbenchmarks to obtain a breakdown of the cost in CPU cycles associated with a switch. Results are presented in Figure 6. Compartment switches can be broken down into hot and cold switches. This is a result of cache utilization. The vast majority of switches (>99.9%) observed in all configurations fall into the category of hot switches. The cold switches, therefore, represent a worst-case cycle latency for compartment switches. These can be expected in compartmentalization scenarios where compartment switches rarely occur, and cache utilization is worse. This results in a best switching latency of <400 cycles, and a worst case of 900-1000 cycles.

6 Related Work

Compartmentalization. In recent years, many works have looked at implementing various forms of compartmentalization [5–10, 12–16, 18, 22, 25, 27, 28, 32–34, 36–38, 44]. Many of these approaches have focused on library isolation [5, 6, 15, 22, 25, 28, 32–34, 37, 44], while others approach isolation in a much more fine-grained way, including function level isolation [13, 14, 36, 38]. Isolation in single-address-space OSes such as Library OSes has also been explored [22, 33, 36], although using other mechanisms such as memory protection keys.

CHERI. While compartmentalization using CHERI hardware capabilities has been explored in the past [5, 13, 45], little work has been done to explore compartmentalization on ASICs, instead using FPGA prototypes. Cap-VMs [32] provides capability aware VM-like abstractions which can be used to isolate capability-unaware components of a system using CHERI RISC-V. CHERI JNI [11] uses capabilities to provide memory safety for the Java VM. Finally, CheriBSD [41] uses capabilities to enable (manual) application compartmentalization.

7 Conclusion

We have proposed two solutions to implement compartmentalization on top of a real-world hardware capability-enabled SoC, ARM Morello. Each solution represents a specific point in the compartmentalization design space, and we have explored the trade-offs they represent in terms of engineering effort for retrofitting legacy software, performance overheads, security, and scalability. We show that hardware capability-based compartmentalization can be achieved with a variable engineering cost, which can be quite low if one is willing to trade-off on scalability and security, and that performance overheads are similar to other intra-address space isolation mechanisms (e.g. memory protection keys), and lower than more heavyweight (page table/extended page table) solutions.

8 Acknowledgments

References

[1] Libsodium website, 2023. https://doc.libsodium.org/.

[2] Sqlite website, 2023. https://www.sqlite.org/index.html.

[3] Ioannis Agadakos, Manuel Egele, and William K. Robertson. Polytope: Practical memory access control for C++ applications. *CoRR*, abs/2201.08461, 2022.

[4] Defence Advanced Research Projects Agency. Broad agency announcement compartmentalization and privilege management (cpm).

[5] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Compartos: CHERI compartmentalization for embedded systems. *CoRR*, abs/2206.02852, 2022.

[6] Markus Bauer and Christian Rossow. Cali: Compiler-assisted library isolation. In Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung, editors, *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 550–564. ACM, 2021.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into Reduced-Privilege compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, April 2008. USENIX Association.

[8] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[9] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 193–204, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long 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*, pages 56–71. IEEE Computer Society, 2016.

[11] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert M. Norton, Stacey D. Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N. M. Watson. CHERI JNI: sinking the java security model into the C. In Yunji Chen, Olivier Temam, and John Carter, editors, *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*, pages 569–583. ACM, 2017.

[12] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206, 2015.

[13] Lawrence G. Esswood. CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor. Technical Report UCAM-CL-TR-961, University of Cambridge, Computer Laboratory, September 2021.

[14] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1016–1031. ACM, 2015.

[15] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC'19. USENIX Association, 2019.

[16] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.

[17] Paul 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*, pages 32–37. IEEE Computer Society, 1987.

[18] Douglas Kilpatrick. Privman: A library for partitioning applications. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association.

[19] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. CCS '13, page 721–732, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Hugo Lefeuvre, Vlad-Andrei Badoiu, Yi Chen, Felipe Huici, Nathan Dautenhahn, and Pierre 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.

[22] Hugo Lefeuvre, Vlad-Andrei Badoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. Flexos: towards flexible OS isolation. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 467–482. ACM, 2022.

[23] Hugo Lefeuvre, Vlad-Andrei Badoiu, Stefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Razvan Deaconescu, Felipe Huici, and Costin Raiciu. Flexos: making OS isolation flexible. In Sebastian Angel, Baris Kasikci, and Eddie Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, pages 79–87. ACM, 2021.

[24] ARM Limited. Arm Morello System Development Platform (SDP) Technical Reference Manual, 2022.

[25] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS'17. Association for Computing Machinery, 2017.

[26] Arm Ltd. Arm® architecture reference manual supplement morello for a-profile architecture. Technical report, 2022.

[27] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E. Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 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.

[28] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20. USENIX Association, 2020.

[29] Robert M Norton. Hardware support for compartmentalisation. Technical report, Cambridge, UK, 2016.

[30] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

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

8

[32] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 597–612, Carlsbad, CA, July 2022. USENIX Association.

[33] Vasily A. Sartakov, Lluís Vilanova, and Peter R. Pietzuch. Cubicleos: a library OS with software componentisation for practical isolation. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 546–558. ACM, 2021.

[34] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20. USENIX Association, 2020.

[35] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Osdi*, volume 10, pages 33–46, 2010.

[36] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19. USENIX Association, 2019.

[38] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 644–662, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Robert Watson. The arm morello board, 2022. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html.

[40] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016.

[41] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 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*, pages 20–37. IEEE Computer Society, 2015.

[42] Robert NM Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, et al. Cheri c/c++ programming guide. Technical report, Cambridge, UK, 2020.

[43] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, 2014.

[44] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Proceedings of the 17th European Symposium on Research in Computer Security*, pages 859–876. Springer Berlin Heidelberg, 2012.

[45] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, et al. Cherirtos: A capability model for embedded devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 92–99. IEEE, 2018.

9