Evaluation of performance and security strengths of library–based compartments created on Morello Boards: technical report

Regis Schuch[1], **Rafael Z. Frantz**[1], **Carlos Molina-Jimenez**[3]

[1]Unijuí University – Ijuí – Brazil

[2]Computer Lab, University of Cambridge

{regis.schuch, rzfrantz}@unijui.edu.br,

carlos.molina@cl.cam.ac.uk

Abstract. *This report evaluates compartments created using the library–based compartmentalisation tool available on Morello Boards running the cheriBSD 24.5 operating system. It evaluates the performance costs incurred by the compartments and the strengths of the memory isolation that they provide. It provides links to the Git repositories that store the C and Python codes used in the evaluation and the metrics collected in CSV files. It also includes the plots of the results, a discussion of our interpretation and detailed instructions to encourage practitioners to repeat our experiments and compare their results against ours.*

1. Introduction

It is widely documented that a large percentage of successful attacks are based on memory corruption techniques. In response, several techniques have been devised to protect memory that some authors group them into two large classes: exploit mitigation and compartmentalisation [Watson et al. 2015]. Approaches of the first class deploy techniques (for example patches) to prevent the occurrence of known attacks, in this sense, they are corrective techniques. Compartmentalisation Approaches account for attacks never seen before, as such, they are more general. We can say that compartmentalisation's aims is to mitigate the class of attacks that exploit memory vulnerabilities.

Intuitively, the idea is to build *cages* in memory to execute code under strict control to stop compromised code from doing bad things (technically, from executing illegal operations) outside the cage.

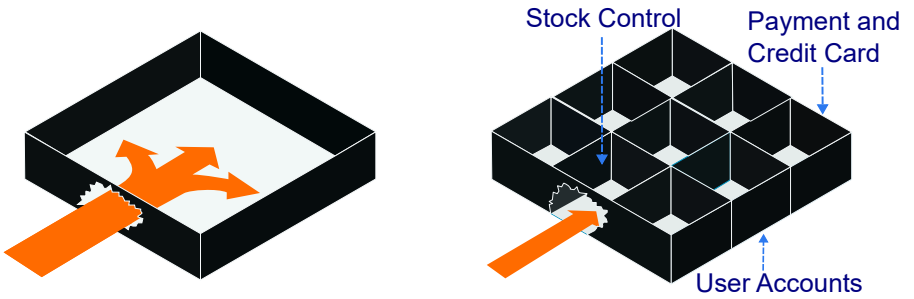
More technically, the idea to divide large complex software into components or modules and to run each component in isolation under the least privilege principle. As a result, the attacking surface and the impact of successful attacks is reduced.

Examples of large software are operating systems, kernels, web browsers and also, user applications composed of several modules implemented separately such as applications that use libraries implemented by third parties. In this work we use user applications in our experiments.

An example of application that can benefit from compartmentalisation is digital payment. A digital payment service can be divided into separate modules (for example, credit card data, user account management, authentication, etc.) to be run separately, each in its own compartment. In this manner the consequences of successful attacks on a components does not spread to other components.

In executions under the least privilege principle, each component is granted access only to the resources needed to accomplish its task, crucially, it is granted access only to the memory region needed to run.

The salient advantage of compartmentalisation is that it is more general than the mitigation of known attacks. Compartmentalisation accounts for known and unknown exploit techniques [Watson et al. 2015]; it assumes that both kinds of attacks can potentially succeed, therefore it aims at mitigation through reduction of the attack surface and spread and spread and propagation of the consequences. With compartmentalisation, corruption of a component of the application affects only the resources associated to the component rather than the whole application. The impact of a successful attack on an application implemented with and without compartments if shown graphically in Fig. 1.



(a) **Breach** has full access (b) **Breach** is contained to a specific area

Figure 1. Attack impact: a) without and b) with memory compartmentalisation (adapted from [ARM 2019]).

Different terminology is used to refer to compartments, including, sandboxes and security domains. Different approaches has been used to implement compartments that guarantee different security properties [Watson et al. 2015, Watson 2019b].

Compartment are central the CHERI (Capability Hardware Enhanced RISC Instructions) project [Watson 2019a] where they are regarded as a general solution to security problems and implemented with cheri–capabilities.

A cheri-enabled computer is a computer with a standard ISA (for example Arm's arch64) extended with additional ISA operations to implement cheri-capabilities and to manipulate them. A cheri–capability (cheri-cap, capability or simply a cap) is a conventional memory pointer extended with additional parameters (tags, bounds, permissions and sealing) to guarantee that operations that involve capabilities observe some security properties. To manipulate these capabilities the ISA has been extended with additional ISA operations that guarantee the observance of two fundamental properties in capability manipulation: provenance (they are derived, through a legal path, from the root of trust at boot time) they are monotonic (no entity is able to create a capability with more permissions).

to reorganise and improve.

An example of property is that the capability will never point outside a given memory region. Another example is that a piece of code can access a given memory address for reading, only if it has a capability with reading permissions and able to point to that address.

Cheri capabilities are able to operate with virtual memory addresses and are meant to complement the protection provided by Memory Management Units (MMUs).

As demonstrated by the Morello Board and Sonata Board [DSbD programme 2024], cheri-caps can be used to implement a variety of software-layer models pursuing different software operational models. To this end, the latest release of cheriBSD includes on–going work of two different CHERI-enabled software compartmentalisation models [Watson 2019b, Watson and Davis 2024]

- Collocated processes (co–processes).

- Library–based compartmentalisation.

Co–processing is the execution of several related processes in the same address space to ease their interprocess communication. Capabilities are used to prevent accidental and malicious interprocess interaction. This approach is at an earlier stage of development; therefore, we will not discuss it further [Watson and Davis 2024].

One of the appealing features of library–based compartmentalisation is that it enables the programmer to create compartments seamlessly, that is, transparently, rather than explicitly using low level instructions. Yet, it is currently under development, as such, there is no practical evidence of the performance cost that compartments created with this tool incurs or of their security strengths.

To help clarify the question, we have conducted several experiments with library–based compartments available on Morello Boards running the cheriBSD 24.5 operating system. This report documents the experiments and results.

2. Library–based compartmentalisation

Library–based compartmentalisation is a programming model where each module (for example a dynamic library) of the program is executed in a separate compartment which are considered independent trust domains. Transitions between domains (forwards and backwards) is controlled by a trampoline function generated and inserted by the dynamic linker [Gao and Watson 2024]. Both the linker and trompoline run in user space but in executive mode and therefore are allowed to read and write restricted mode registers.

The tools have been implemented to help programmers to execute programs composed of several modules (for example, dynamic libraries) using compartmentalisation.

Library-based compartmentalisation uses the dynamic linker [Bartell et al. 2020] to shield the complexity of compartment creation from the programmer and is assumed to belong to the Trusted Computing Base (TCB) of the Morello Board (see Section 2.0.1).

As shown in several examples demonstrated in subsequent sections, the programmer only needs to specify some flags at compilation and execution time to request execution with compartmentalisation.

When a dynamically linked program is launched, the dynamic linker locates, loads, and binds the libraries to the program. It performs symbol resolution to connect function calls found in the program to their definitions in shared libraries. It uses the Procedure Linking Table (PLT) and Global Offset Table (GOT) to manage indirect references and dynamically update symbol addresses.

The responsibility of the trampoline function is manages function calls across compartments. It mediates to adjust registers and stack pointers to guarantee compartment integrity [Gao and Watson 2024, Connolly 2024].

The operation of the dynamic linker and the trampoline function is illustrated in Fig. 2. The figure shows the execution of an application that creates a parent and a child process the communicated with each other through a pipe. The dynamic linker creates and updates the trampoline function. They are in yellow boxes to indicate that both belong to the Trusted Computing Base. As mentioned above, both run in user space but in executive mode, therefore, neither the parent or the child library has capabilities to manipulate the linker or the trampoline function.

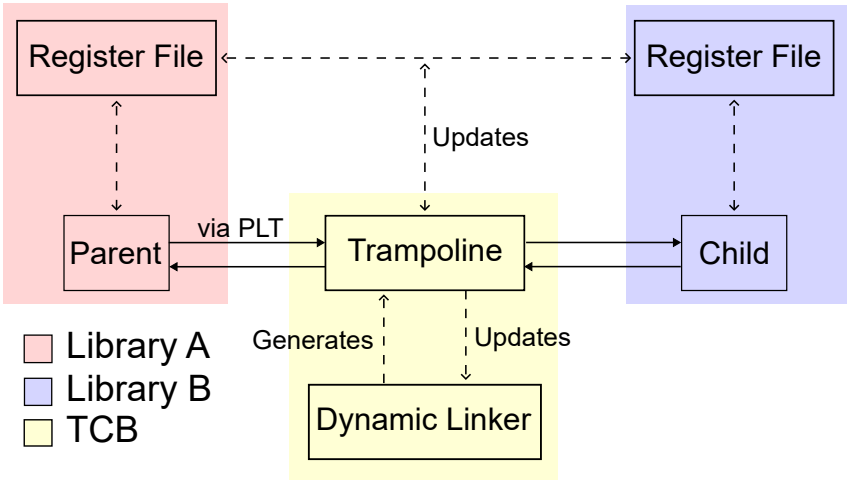


Figure 2. Dynamic linker generating a trampoline function between a parent and child process.

The parent and child process are regarded as libraries and as such are allocated to separate compartments (library A in pink and library B in purple, respectively) with their own register files. At run time, the trampoline function operates as an intermediary: it intercepts and redirects calls.

2.0.1. Trusted computing base

In the cheri stack, cheri–caps are created and transferred following the provenance validity and monotonicity properties [Watson et al. 2020]. Provenance demands that cheri–caps are derived via valid transformations of valid capabilities. On the other hand, monotonicity required that the permissions of a given capability does not exceed the permissions of its creator. These properties are strict and followed during the boot time process and by user applications [Watson et al. 2020]. The boot procedure shows that trust in the Morello Board is rooted on the firmware which plays the role of the root of trust [Cofta 2007].

1. At boot time, a cheri enabled platform provides initial capabilities to its firmware. The latter can them access data and fetch instructions from the full address space.
2. The firmware clears all capability tags from memory.
3. The firmware derives and transfers capabilities to the boot loader.
4. The boot loader derives and transfers capabilities to the hypervisor.
5. The boot hypervisor derives and transfers capabilities to the operating system.
6. The operating derives and transfers capabilities to user applications.
7. User applications can derive capabilities for internal use (for example, to allocate memory) and for transferring to their modules.

The monotonicity property is strictly observed at each stage. Also, at each stage, bounds and permissions may be restricted to further limit access permissions of the receiver of the capability. For example, the OS may assign capabilities for only a limited portion of the address space to a given user application, say, to run it within a compartment.

3. Experiments set up

We use a Morello Board, which is physically located in Toronto, within the premises of TODAQ[1], a non-funding partner of the CAMB project[2]. As shown in Figure 3, a laptop connected to the network of the Applied Computing Research Group (GCA)[3] at Unijuí, Brazil, is used to access the Morello Board via an ssh connection. The figure outlines the main configuration parameters of the Morello Board, while Table 1 lists additional parameters and the CheriBSD commands needed to output these configurations directly from the board.

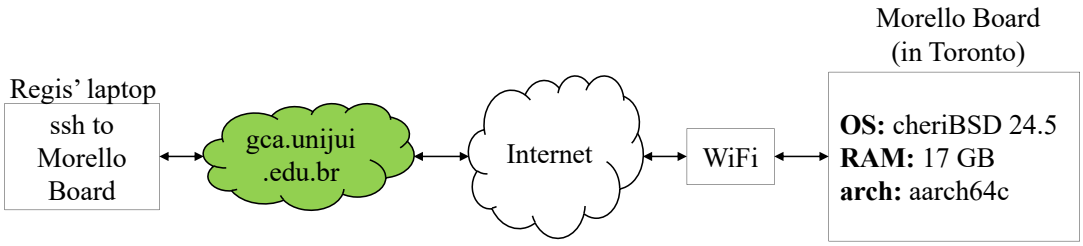


Figure 3. Morello Boards location.

We specify the hardware and software configurations of the Morello Board used in the experiments in Table 1.

It is worth explaining that, as shown in the csv files available from Git, in our experiments we repeated the execution of each operation 100 times, collected the measurements and averaged the results. The choice of 100 repetitions was based on the Central Limit Theorem, which suggests that a sample size of 100 is often adequate to yield a statistically meaningful average [Statistics How To 2023].

3.1. Compilation and execution

The inclusion or exclusion of library based compartments is determined at compilation and execution time as documented in the manual [Gao 2024, Cheri Team 2022, Watson 2019b].

3.1.1. Compilation and execution without library-based compartments

The normal compilation (without the inclusion of library-based compartments) is shown in the following example for a `helloworld.c` program.

```
$ clang-morello -o hello hello.c
```

To execute `helloworld`, the programmer can type:

```
$ ./helloworld
```

3.1.2. Compilation and execution with library-based compartments

The following command shows the compilation flags to enable library-based compartments:

[1] `https://engineering.todaq.net/`
[2] `https://www.cl.cam.ac.uk/research/srg/projects/camb/`
[3] `http://gca.unijui.edu.br/`

Table 1. Morello board configuration parameters used in the experiments and the online cheriBSD commands to output them.

Component	Specification	Command
Operating System	CheriBSD 24.5 (FreeBSD 15.0-CURRENT)	`uname -a`
Kernel Version	FreeBSD 15.0-CURRENT, releng/24.05	`uname -v`
Board	Morello System Development Platform	`kenv | grep smbios.system.product`
RAM	17 GB detected (16 GB DDR4, 2933 MT/s, ECC)	`dmidecode --type memory`
Storage	SSD	`camcontrol identify ada0`
Architecture	aarch64c (with CHERI support)	`sysctl hw.machine_arch`
Processor Model	Research Morello SoC r0p0	`sysctl hw.model`
Number of CPUs	4	`sysctl hw.ncpu`
Compiler	clang (with Morello support)	`clang-morello --version`
Tool	proccontrol (for CHERI compartments)	`proccontrol -m cheric18n -s enable ./binary`
Python	Python 3 (required for Experiments 1, 5 and 6)	`python3 --version`
Scripts used	cheri-cap-experiment.py, cpu-in-experiment.c, memory-in-experiment.c, pipe-in-experiment.c, pipe-trampoline-in-experiment.c, library_a.c, library_b.c, memory_reader.py, integration_process.c	Not applicable
Access	Remote via SSH	`ssh -i private_key user@server`

```
$ clang-morello -march=morello+c64 -mabi=purecap -o helloworld
   helloworld.c
```

The `-march=morello+c64` parameter defines the 64-bit Morello architecture, and `-mabi=purecap` sets the ABI (Application Binary Interface) for the secure environment, representing all memory references and pointers as capabilities.

To execute the executable `helloworld` in a library-based compartment, the programmer can type:

```
$ proccontrol -m cheric18n -s enable helloworld
```

The binary was executed with library compartmentalisation enabled using `proccontrol`.

We use the example shown above in subsequent sections to compile and execute the programs used in the evaluation.

4. Evaluation of the max number of library–based compartments

The main aim of this experiment is to measure and analyse how the memory of a Morello Board is consumed by instances (also called replicas) of attestables. To this end, we create and attestable and load it with a C program compiled with the library compartmentalisation tool. We use the enterprise application integration (see yellow box) use case implemented in - `tee-compartimentalisation-study-case` repository[4].

The parameter to measure is the number of attestables that can be created on a Morello Board before consuming 90% of its memory. In addition to the number of attestables, we took the opportunity to collect metrics about the time it takes the operating system to wipe the memory used by the attestable. The set up o the experiment is shown in Fig. 4.

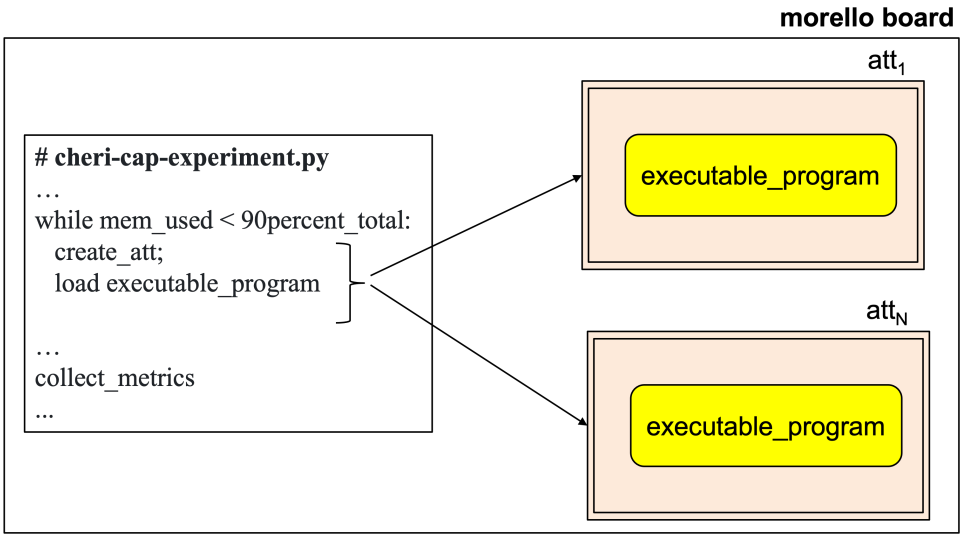


Figure 4. Max number of attestable that can be created before exhausting memory.

Imagine that user Alice is conducting the experiment. To create the attestables and collect the metrics, Alice executes the following steps:

1. Initiation: Alice initiates cheri-cap-experiment.py on a Morello Board.
2. Launch: Alice executes cheri-cap-experiment.py to launch the attestable.

   `% cheri-cap-experiment.py`[5]
3. % python3 cheri-cap-experiment.py runs incrementally creating attestable replicas until it detects that the attestables have consumed 90% of the 17118.4 MB of the Morello Board's memory, that is, about 15406.5 MB.

4.1. Results

The results are logged in the CSV file `cheri-cap-experiment-results.csv` which is available from Git `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/cheri-caps-executable-performance/cheri-cap-experiment-results.csv`.

[4]Repository available at: `https://github.com/gca-research-group/tee-compartimentalisation-study-case`

[5]`https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/cheri-caps-executable-performance/cheri-cap-experiment.py`

The first few lines of the CSV file are shown in Table 2 to be read as follows:

The table contains the following measurements:

Number of Compartments: The number of compartments created.

Memory Used (MB): The amount of memory consumed by the given number of compartments.

Time Elapsed: The time elapsed since the beginning of the experiment that is assumed to start at time zero.

Let us assume that the experiment stars at time zero, with 0 number of compartments which has consumed zero MB of memory.

The first row shows that it took 514.00 ms to `cheri-cap-experiment.py` to create one compartment that consumes 1628.40 MB of memory. As a second example take the 5th row. It shows that after 10808.39 ms, `cheri-cap-experiment.py` has created 5 compartments that have consumed 1640.39 MB.

Table 2. Metrics of memory consumed by different numbers of attestables and elapsed time.

Number of Compartments	Memory Used (MB)	Time Elapsed (ms)
1	1628.40	514.99
2	1631.00	3070.37
3	1634.03	5656.81
4	1637.11	8222.68
5	1640.39	10808.39
...	...	...
8991	13066.42	26773287.54

The blue line in the plot of Fig. 5 illustrates how memory is consumed as the number of compartments increases. The orange line illustrates the elapsed time as the number of compartments increases.

We initially expected memory consumption to increase steadily from 1,628.3 MB, corresponding to a single attestable replica, to 15,406.5 MB (90% of total memory) consumed by N attestable replicas. The objective was to determine the exact value of N.

However, the results revealed unexpected behaviour: memory consumption increased consistently only until approximately 3,800 attestable replicas consumed 14,582.5 MB. After this point, memory consumption began to decrease as the number of attestable replicas continued to rise. The final data point shows that 8,991 attestable replicas consumed 13,066.4 MB, or roughly 76% of the total memory.

We did not expect the behaviours exhibited by the blue line of Fig. 5. We have no sound explanation for it. These preliminary results highlight an area for further exploration. Additionally, the analysis of the time required to wipe the memory of the attestable replicas remains pending.

5. Memory performance in the execution of allocate, release, read and write operations

To collect metrics we execute a C program compiled and executed without compartments and with compartments:

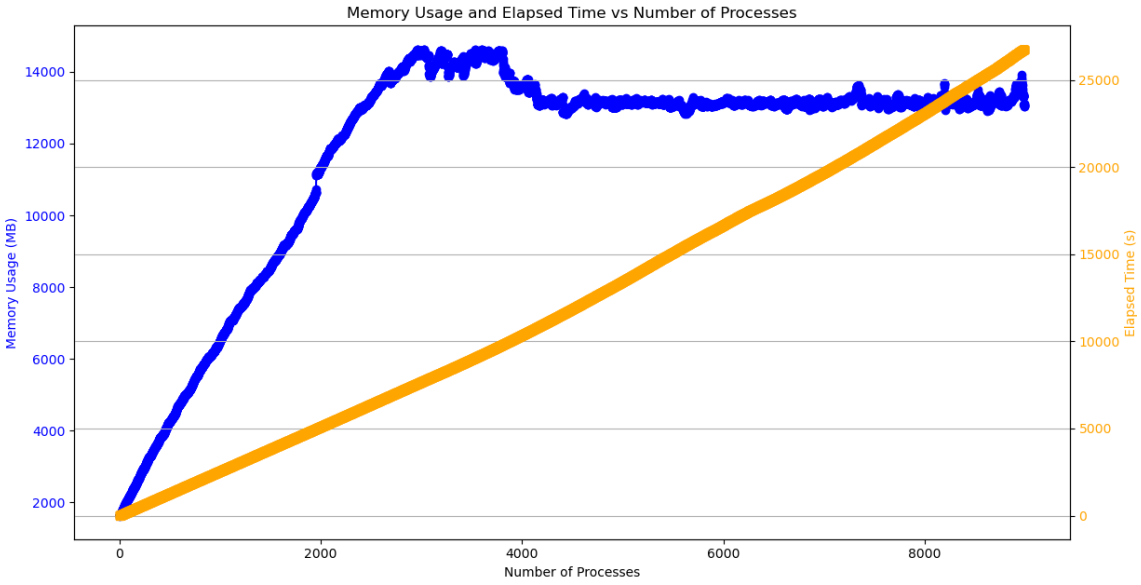


Figure 5. Memory consumed by incremental replication of compartments and time to create compartments.

- **Compilation and execution without compartments:** `memory-out-experiment.c`[6].
  We have compiled and execute it with the following cheriBSD commands:

```
$ clang-morello -o memory-in-experiment memory-in-
   experiment.c -lm

$ ./memory-in-experiment
```

- **Compilation and execution with compartments:** The `memory-in-experiment.c`[7].

```
$ clang-morello -march=morello+c64 -mabi=purecap -o memory
   -in-experiment memory-in-experiment.c -lm

$ proccontrol -m cheric18n -s enable memory-in-experiment
```

In this experiment, we use the code shown in Algorithm 1. It executes the following operations on large blocks of memory:

a) **allocation:** time required to allocate a block of memory.
b) **write:** time required to write data to fill the entire memory block.
c) **read:** time taken to read the data from the entire memory block.
d) **free:** time taken to release the memory block back into the main memory.

[6] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/memory-performance/outside-tee-exection/memory-out-experiment.c`
[7] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/memory-performance/inside-tee-execution/memory-in-experiment.c`

As shown in Fig. 6, we use blocks of $100, 200, 300, ..., 100000MB$ as large blocks of memory. Blocks of these sizes are typical of applications that process images and access databases.

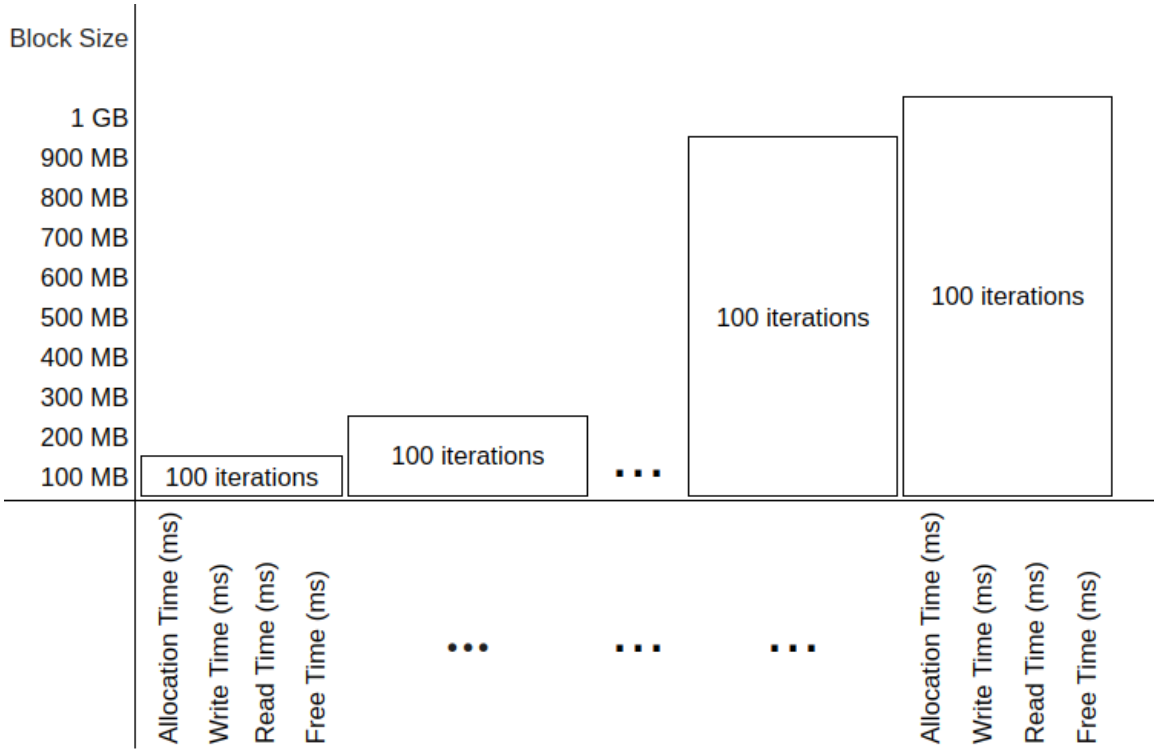


Figure 6. Performance of memory operations on memory blocks of different sizes.

Execution begins with the `perform_trials` function (line 1), which receives a results file as input parameter to store performance metrics including the total time taken to run the trials. The for–loop (line 3) iterates over memory blocks of different sizes ranging from `MIN_BLOCK_SIZE` to `MAX_BLOCK_SIZE` with increments specified by `BLOCK_STEP`. The inner for–loop (line 4) repeats the trial `NUM_TRIALS` times for each block size. `NUM_TRIALS` is defined by the programmer as a constant.

At each iteration, the memory allocation duration is measured with the time function (line 5); the time to write to the block is measured in line 6, the time to read the block is measured in line 7, and, finally, the time to free the memory is measured in line 8. The metrics collected are recorded in the results file along with the trial number (line 9).

5.1. Results

The metrics collected are stored in two separate CSV files: `memory-in-experiment-result.csv`[8] for the run inside a compartment. The file `memory-out-experiment-result.csv`[9] collects metrics of the run without compartments. We calculate the average time that it takes to allocate, write, read and free for each block size of 100 MB, 200 MB, 300 MB, etc.).The results are summarised in Tables 3 and 4.

[8] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/memory-performance/inside-tee-execution/memory-in-experiment-resuls.csv`

[9] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/memory-performance/outside-tee-exection/memory-out-experiment-resuls.csv`

Algorithm 1 Execution of memory operations and metric collections of their executions.

```
perform_trials(results_file, total_execution_time)
begin
for current_block_size in MIN_BLOCK_SIZE to MAX_BLOCK_SIZE step BLOCK_STEP do
    for trial_number from 1 to NUM_TRIALS do
        allocation_duration = time(allocate_memory(current_block_size))
        write_duration = time(write_to_memory(memory_block, current_block_size))
        read_duration = time(read_from_memory(memory_block, current_block_size))
        free_duration = time(deallocate_memory(memory_block))
        log(results_file, current_block_size, trial_number, allocation_duration, write_duration, read_duration, free_duration)
    end for
end for
end
```

Table 3. Metrics of runs inside a compartment, including mean and standard deviation.

Block Size (MB)	Allocation Time (ms)	Write Time (ms)	Read Time (ms)	Free Time (ms)
100	93 ± 171.27	283,239 ± 58.31	283,133 ± 28.83	89 ± 180.05
200	98 ± 221.17	566,458 ± 82.10	566,269 ± 65.02	214 ± 397.35
300	99 ± 295.44	849,705 ± 131.43	849,396 ± 87.16	222 ± 452.92
400	127 ± 430.92	1,132,983 ± 189.58	1,132,550 ± 106.44	430 ± 788.02
500	159 ± 599.09	1,416,190 ± 189.97	1,415,698 ± 123.68	217 ± 420.54
600	151 ± 648.00	1,699,454 ± 255.41	1,698,795 ± 174.82	439 ± 921.59
700	195 ± 880.05	1,982,654 ± 245.07	1,981,909 ± 122.70	453 ± 979.92
800	216 ± 1,084.49	2,265,901 ± 235.38	2,265,075 ± 139.94	818 ± 1,513.98
900	288 ± 1,536.92	2,549,115 ± 258.37	2,548,205 ± 196.83	816 ± 1,579.74
1000	248 ± 1,543.50	2,832,372 ± 337.74	2,831,332 ± 167.56	444 ± 1,003.29

- **Allocation time:** A comparison of Table 3 against Table 4 reveals that it takes longer to allocate memory blocks inside compartments. For example, the allocation of 100 MB takes 2 ms when no compartment is used. In contrast, it takes 93 ms when the allocation is requested within a compartment. Another observation is that the time to allocate blocks of memory of different sizes (from 100 to 1000 MB) of runs without a compartment varies from 2 to 5 ms. In contrast, the time to allocate memory within a compartment varies significantly from 93 to 288 ms and depends on the size of the block. Times range from 93 ms for 100 MB blocks to 288 ms for 900 MB blocks. In contrast, the time to allocate memory without compartments is shorter, ranging from 2 to 5 ms for all block sizes.
- **Write time:** Both tables show a linear increase in write time as the block size increases. However, the execution inside a compartment takes longer. The difference becomes more evident when the sizes of the blocks increase.
- **Read time:** The time to execute read operations increases linearly in both executions. However, the execution within a compartment takes longer than the execution without compartments.

Table 4. Metrics of runs outside a compartment, including mean and standard deviation.

Block Size (MB)	Allocation Time (ms)	Write Time (ms)	Read Time (ms)	Free Time (ms)
100	2 ± 4.77	282,584 ± 13.86	282,581 ± 12.79	6 ± 4.52
200	4 ± 4.19	565,164 ± 17.12	565,163 ± 18.85	10 ± 4.03
300	4 ± 1.77	847,755 ± 21.18	847,752 ± 64.89	13 ± 3.66
400	5 ± 3.09	1,130,330 ± 21.00	1,130,328 ± 28.20	14 ± 2.27
500	5 ± 3.07	1,412,907 ± 31.49	1,412,903 ± 28.92	15 ± 2.37
600	5 ± 1.56	1,695,493 ± 32.97	1,695,493 ± 30.19	16 ± 1.28
700	5 ± 1.52	1,978,083 ± 52.24	1,978,098 ± 79.47	17 ± 0.86
800	5 ± 1.73	2,260,662 ± 41.09	2,260,660 ± 53.11	18 ± 0.62
900	5 ± 0.54	2,543,249 ± 47.19	2,543,234 ± 42.16	18 ± 0.97
1000	5 ± 0.50	2,825,823 ± 47.72	2,825,818 ± 41.68	18 ± 0.64

- **Free time:** The metrics in the tables show contrasting performances. Table 3 shows that it takes significantly longer to free memory in executions inside a compartment. The times range from 89 to 818 ms. In contrast, Table 4 shows times that range from 6 to 18 ms in executions without compartments.

Plots of the results from Tables 3 and 4 shown in Fig. 7. Full records are available from `memory-in-experiment-result.csv` and `memory-out-experiment-result.csv`.

A graphical view of the results shown in Table 3 and Table 4 is presented in the boxplots of Fig. 8. The dispersion of the time to execute allocate and free operation within a compartments and without compartments is striking.

6. CPU performance in the execution of demanding arithmetic operations

We have carried out this experiment to determine if library–based compartments affect the performance of the CPU. Precisely, we have executed a program with functions that involve the execution of CPU–demanding arithmetic operations and collected metrics about execution time. The program that we have implemented for this purpose, includes operations with integers (int), floating point (float), arrays, and complex mathematical functions (such as trigonometric and exponential functions) that are known to be CPU–demanding.

We use a C program that compile and run inside a library-based compartment and without compartments.

- **Compilation and execution inside a compartment**
  The program that we use is available from Git: `cpu-in-experiment.c`[10]
  We compile and run it as follows:

```
$ clang-morello -march=morello+c64 -mabi=purecap -o cpu-in
   -experiment cpu-in-experiment.c -lm

$ proccontrol -m cheric18n -s enable cpu-in-experiment
```

[10] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/cpu-performance/inside-tee-execution/cpu-in-experiment.c`

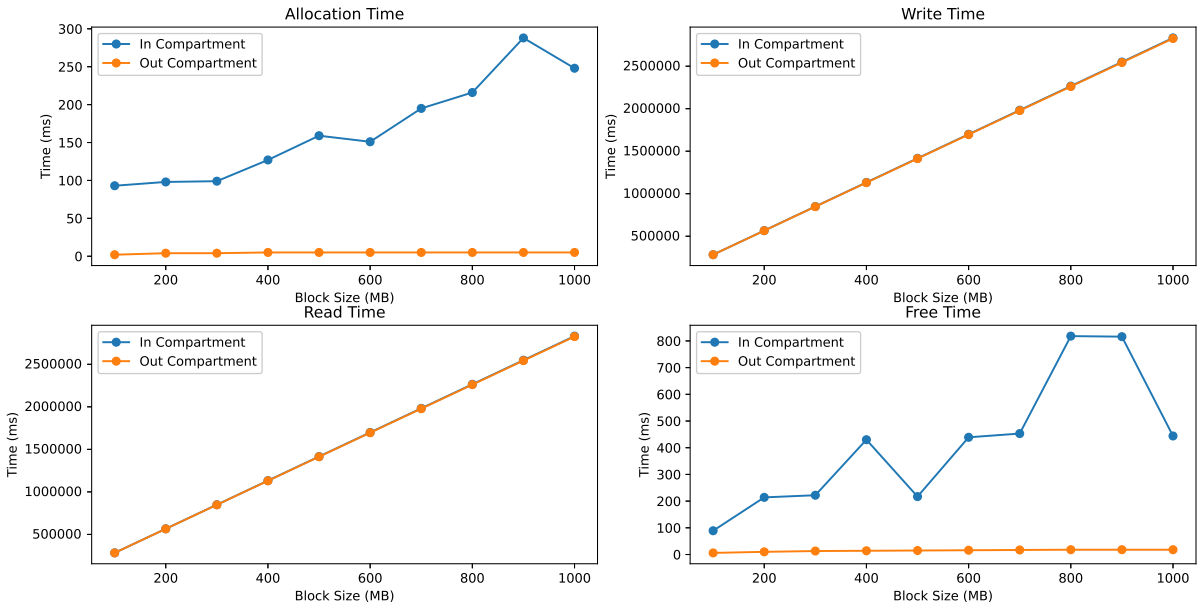


Figure 7. Time to execute allocate, write, read and release memory operations.

- **Compilation and execution without a compartment**
  The program that we use is available from Git:
  `cpu-out-experiment.c`[11]
  We compile and run it as follows:

```
$ clang-morello -o cpu-out-experiment cpu-out-experiment.c
    -lm

$ ./cpu-out-experiment
```

The choice of these operations is based on the variety of typical workloads in computer applications, covering operations that vary in CPU resource usage. Time collection was carried out in both environments, allowing a detailed comparison between performance in the compartmentalised environment and the Morello Board's normal operating environment.

Algorithm 2 contains the code that we have run to produce metrics about the CPU performance and store them in a CSV files.

The execution begins with the `perform_trials` function (line 1), which receives the name of as a log file as input parameter to be used to store metrics about the execution of individual operations and the total time to complete the program. The function enters a repeat loop that is repeated the number of times specified by `num_of_trials` (line 3), where each iteration represents a test identified by `trial_num`. In each iteration, the initial test time is recorded (line 4), followed by the execution of the computational operations determined by `WORKLOAD_SIZE` (line 5). At the end of execution, the final time is recorded (line 6), and the total CPU time elapsed is calculated by subtracting the `start_time` from the `end_time` (line 7). This time is recorded in the log file, along with the test number (line 8), and added to `total_execution_time`, that accumulates the total time spent on all the tests (line 9).

[11] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/cpu-performance/outside-tee-exection/cpu-out-experiment.c`

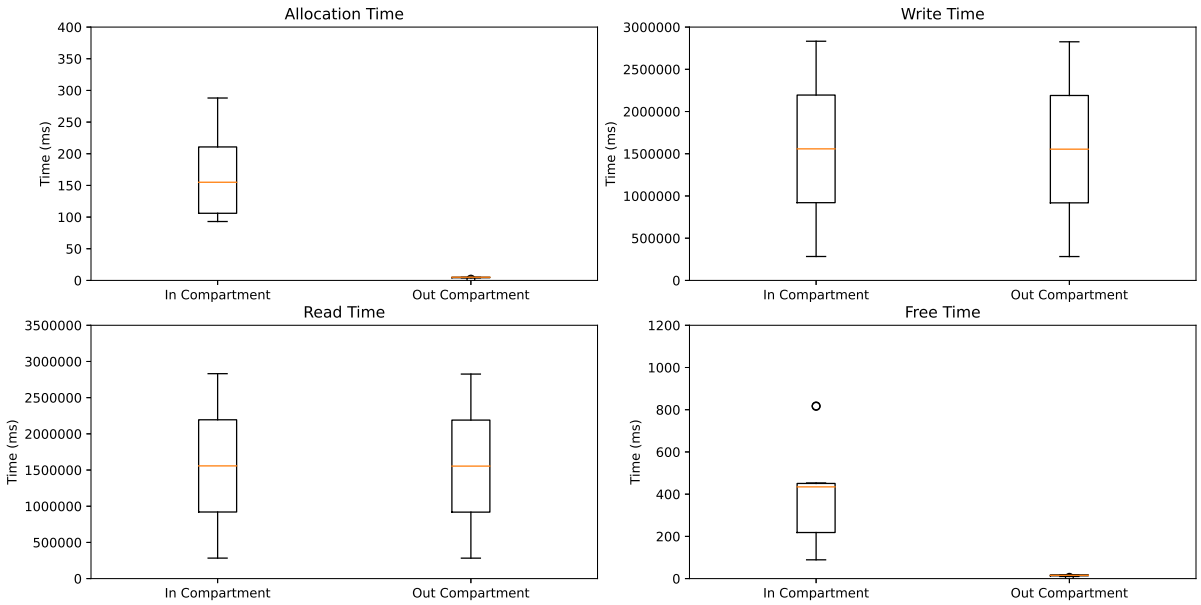


Figure 8. Dispersion of the time to execute allocate, write, read and free perations.

Algorithm 2 CPUPerformance

```
perform_trials(log_file, total_execution_time)
begin
for trial_num in num_of_trials do
    start_time = capture_time()
    execute_operations(WORKLOAD_SIZE)
    end_time = capture_time()
    cpu_time = calculate_cpu_time(start_time, end_time)
    log_results(log_file, trial_num, cpu_time)
    total_execution_time += cpu_time
end for
end
```

6.1. Results

The results collected from the execution inside a compartment are available from `cpu-in-experiment-result.csv`[12]. Similarly, the results collected from the execution without a compartment are available from `cpu-out-experiment-result.csv`[13].

Table 5 compares the average execution times of different operations in both executions.

The results in show that complex mathematical operations (trigonometric and exponential functions) executed within a compartment took 70,780 ms on average. In contrast, the execution of the same operations without a compartment took only 46,759 ms. This represents a performance cost of approximately 51.24%. However, the execution of arithmetic operations with integers without a compartment takes 922 ms. This figure is similar to the 993 ms that it

[12] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/cpu-performance/inside-tee-execution/cpu_in-experiment-result.csv`

[13] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/cpu-performance/outside-tee-exection/cpu-out-experiment-result.csv`

Table 5. Times to execute CPU operations inside and without a compartment.

Trial Type	CPU Time (ms) - Normal	CPU Time (ms) - Secure
Maths (trigon. and exp. func)	46,759	70,780
Int	922	993
Float	830	804
Array	1,407	1,443

takes to execute the same operation inside a compartment. The difference is only 7.70%. Unexpectedly, the execution of floating point operations inside a compartment took 804 ms, which is slightly lower than the execution without a compartment, which took 830 ms. The difference is 3.13%. Finally, the execution of array manipulation operations took 1,443 ms inside a compartment. This is not very different from the 1,407 ms that it takes to execute the same operation without a compartment; precisely, the difference is only 2.56%.

As visualised in Fig. 9, these results indicate that there is a noticeable performance cost in the execution of complex math operations inside compartments. However, in the execution of int, float and array operations, the performance is similar with and without compartments; strikingly, it is slightly better in the run inside a compartment.

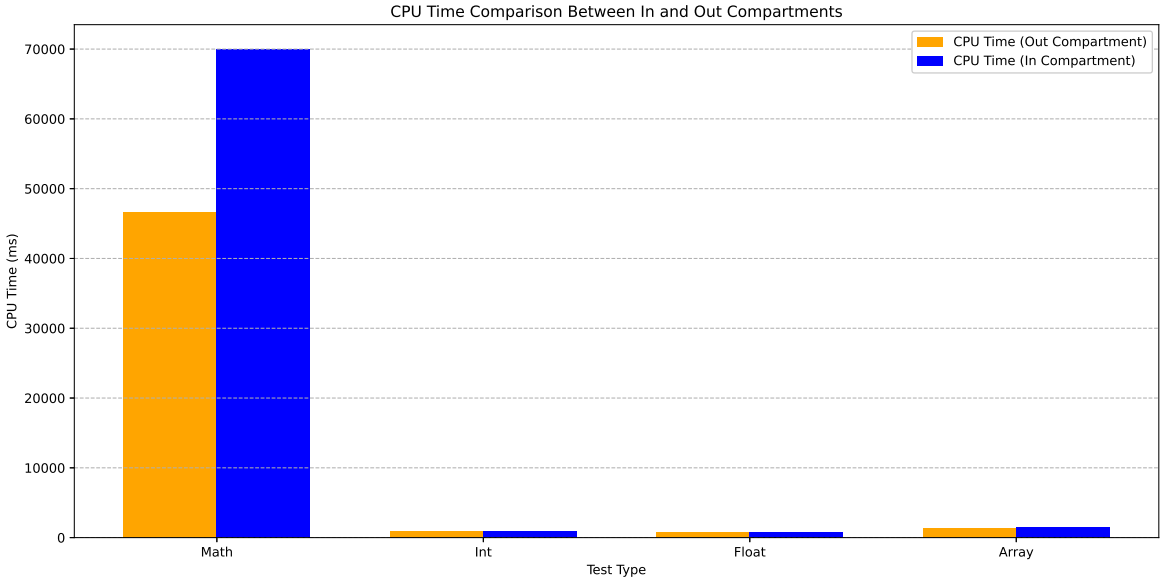


Figure 9. CPU performance in executions within and without compartments.

7. Communication performance over pipes

This experiment was conducted to evaluated how the use of compartments affect the performance of communication over Unix pipes. To collect metrics we have implemented a C program that communicates a parent with child process over a pipe and collects metrics about writing to and reading from a pipe that interconnected them. As shown in Fig. 10, the parent process writes a message to the pipe and the child process reads it.

We run the C program within a compartments [14] and without compartments `pipe-out-experiment.c`[15]

- **Compilation and execution inside a compartment**

```
$ clang-morello -march=morello+c64 -mabi=purecap -o pipe-
   in-experiment pipe-in-experiment.c

$ proccontrol -m cheric18n -s enable pipe-in-experiment
```

- **Compilation and execution without a compartment**

```
$ clang-morello -o pipe-out-experiment pipe-out-experiment
   .c

$ ./pipe-out-experiment
```

To collect metrics, the parent process writes a random string of 1024 bytes —a typical size widely used in inter-process communication applications.

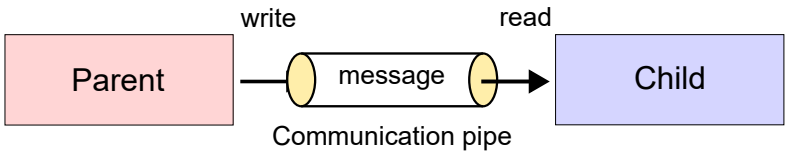


Figure 10. Parent–child communication over a pipe.

We collected metric about the following operations:

a) **write:** time taken to the parent process to write data to the pipe.
b) **read:** time taken to child process to read the data from the other end of the pipe.

The code repeats each operation 100 times. This is in line with the principles of the Central Limit Theorem that states that a larger sample size helps to detect finer fluctuations in latency patterns [Statistics How To 2023].

Algorithm 3 describes the execution of the operations and the settings of timers to collect the metrics.

In Algorithm 3, the `perform_pipe_trial` function (line 1) initiates a sequence of operations that measure the performance of pipe communication between the parent and child processes. The parameters `MESSAGE_SIZE` and `NUM_OF_TRIALS` (lines 3 and 4), establish the message size and the number of messages to be sent, respectively. For each iteration, from 1 to `NUM_OF_TRIALS` (line 5), the parent starts the write timer (line 7), writes a message of size `MESSAGE_SIZE` to the pipe (line 8), stops the write timer (line 9), and then sends the recorded `write_duration` back through the pipe (line 10). The child process, in turn, reads the message and the `write_duration` from the pipe (lines 12 and 13). To collect the metrics, the child process starts the read timer before reading (line 14), and stops it upon completing the reading (line 15). The trial number, along with the write and read durations, is logged in the log file (line 16). The procedure is repeated for each iteration until all messages are written to and read from the pipe (line 17).

[14] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/pipe-performance/inside-tee-execution/pipe-in-experiment-result.c`

[15] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/pipe-performance/outside-tee-execution/pipe-out-experiment-result.c`

Algorithm 3 Pipe Communication Performance

```
perform_pipe_trial(log_file)
begin
define MESSAGE_SIZE
define NUM_OF_TRIALS
for trial_num from 1 to NUM_OF_TRIALS do
    if parent_process then
        start_timer(write_duration)
        write(pipe, message of size MESSAGE_SIZE)
        stop_timer(write_duration)
        write(pipe, write_duration)
    else
        read(pipe, message of size MESSAGE_SIZE)
        read(pipe, write_duration)
        start_timer(read_duration)
        stop_timer(read_duration)
        log_results(log_file, trial_num, write_duration, read_duration)
    end if
end for
end
```

7.1. Results

We stores the data collected from this experiment in two separate CSV files: `ppipe-in-experiment-result.csv`[16] for operations executed inside the compartment and `pipe-out-experiment-result.csv`[17] for operations executed without a compartment.

Table 6 and Table 7 contain the results of each iteration, including message size, write time, read time, and total time taken for the operations.

Table 6. Time to execute write and read from a pipe inside a compartment.

Trial	Message Size (Bytes)	Write Time (ms)	Read Time (ms)	Total Time (ms)
1	1024	0.016	0.161	0.177
2	1024	0.003	0.068	0.071
3	1024	0.003	0.075	0.078
4	1024	0.003	0.077	0.080
...	...	...	...	...
100	1024	0.003	0.079	0.082

The data shows the differences in the performance of inter–process communication (through pipes) inside a compartment and without compartments.

[16] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/pipe-performance/inside-tee-execution/pipe-in-experiment-result.csv

[17] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/pipe-performance/outside-tee-execution/pipe-out-experiment-result.csv

Table 7. Time to execute write and read from a pipe without a compartment.

Trial	Message Size (Bytes)	Write Time (ms)	Read Time (ms)	Total Time (ms)
1	1024	0.013	0.059	0.072
2	1024	0.001	0.001	0.003
3	1024	0.001	0.001	0.002
4	1024	0.001	0.001	0.002
...	...	...	...	...
100	1024	0.001	0.002	0.003

A graphical view of the results is shown in Fig. 11.

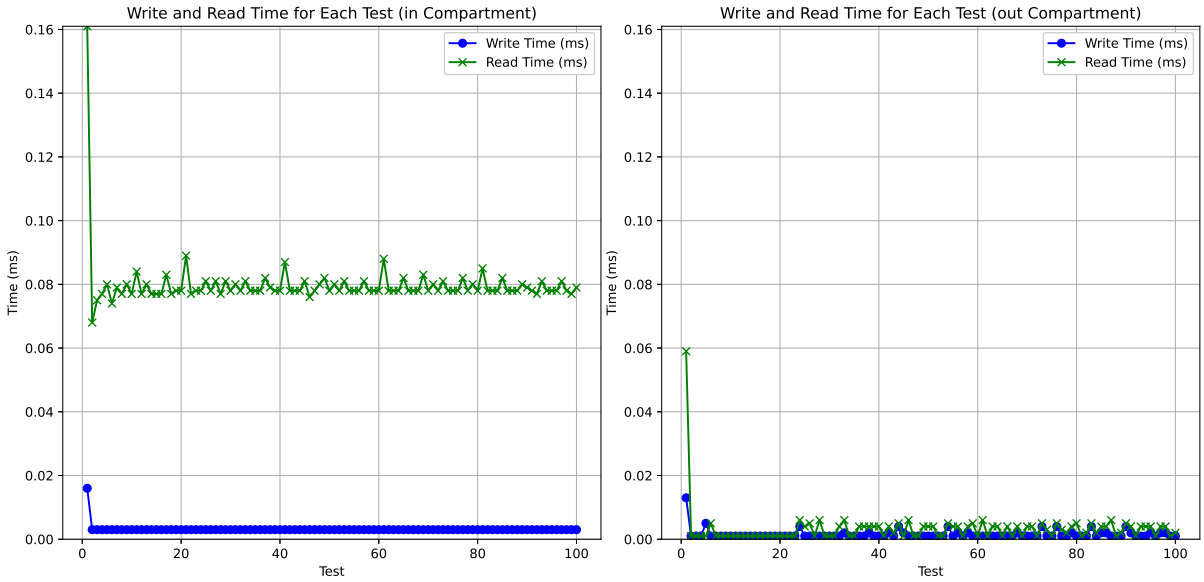


Figure 11. Times to write and read a 1024 byte string from a pipe executed in compartments and without compartments.

The figure reveals that compartments affect performance. The write operation executed inside compartments consistently show a higher latency that ranges from 0.016 ms to 0.003 ms. In contrast, the write time outside compartments is notably shorter, closer to 0.001 ms. This discrepancy highlights the additional computational cost introduced by the compartment.

The effect of the compartment on the performance of the read operation is less severe yet, it is visible. The first test shows a read time of 0.161 ms, compared to 0.059 ms in the execution without compartments. As the tests progress, the execution within the compartment consistently exhibits longer read times. This demonstrates that compartmentalisation introduces delays in inter-process communication.

The results suggest the compartments provide significant benefits in terms of security; yet they incur performance costs; the cost might not be negligible in applications that rely on rapid inter–process communication.

8. Evaluation of Trust Models in Single-Compartment Environments

We have conducted this experiment to examine the trust model that the Morello Board implements. It is documented that the current release of the Morello Board implements an asymmetric trust model where the Trusted Computing Based (TCB) is trusted by the applications but the TCB does not trust the applications. It is worth mentioning that the current Morello Board does not support the mutual distrust model where the privileged software and the applications distrust each other.

To the TCB of the current Morello Board belong the firmware and privileged software that includes the bootloader, hypervisor and operating system. The library-based compartments that we examine in this report, consider that the linker belongs to the TCB too [Gao and Watson 2024].

In this experiment, we use an application written in C `tee-compartmentalisation-study-case` and run it within a compartment and without compartments to examine memory isolation. We followed the following steps:

1. **Compilation and execution:**
   We compiled and executed the application integration within a compartment and without a compartments:
   - Compilation and execution within a compartment:
     The application integration is available from Git: `integration-process-in-experiment.c`[18]
     We compile and run it as follows:

     ```
     $ clang-morello -march=morello+c64 -mabi=purecap -o
        integration_process-in-experiment
        integration_process-in-experiment.c -lssl -lcrypto
         -lpthread

     $ proccontrol -m cheric18n -s enable
        integration_process-in-experiment
     ```

   - Compilation and execution without a compartment:
     The application integration is available from Git:
     `integration-process-out-experiment.c`[19]
     We compile and run it as follows:

     ```
     $ clang-morello -o integration_process-out-experiment
         integration_process-out-experiment.c -lssl -
        lcrypto -lpthread

     $ ./integration_process-out-experiment
     ```

2. **Launch python script:** We launched the Python that performs the memory reading.

   ```
   $ python3 memory_reader.py
   ```

[18] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-single-compartment-performance/inside-tee-execution/integration_process-in-experiment.c

[19] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-single-compartment-performance/outside-tee-execution/integration_process-out-experiment.c

The `memory_reader.py`[20] script cycles through the memory regions of interest reading the data between the start and end addresses of each region directly.

Fig. 12 shows the steps executed by the `memory_reader.py` script:

1. The Memory Reader requests the Cheri OS for the PID of the target process by its name, using the method `getPID(processName)`.
2. Cheri OS returns the corresponding PID.
3. The `memory_reader.py` provides the PID to `getMemoryAddresses(PID)` to request a list of the memory regions associated to the process that have read and write (RW) permissions.
4. CheriBSD responds with the mapped memory regions.
5. The `memory_reader.py` starts scraping the memory directly.
6. For each RW region, it fetches the starting address by calling `seek(startAddress)`.
7. Acknowledgement is return.
8. The `memory_reader.py` executes `read(startAddress to endAddress)` to read the content from the starting address to the end address.
9. The decoded data is return.
   This cycle is repeated for all RW regions.
10. The `memory_reader.py` executes `output(dataReadFromMemory)` to record the data read from the memory.

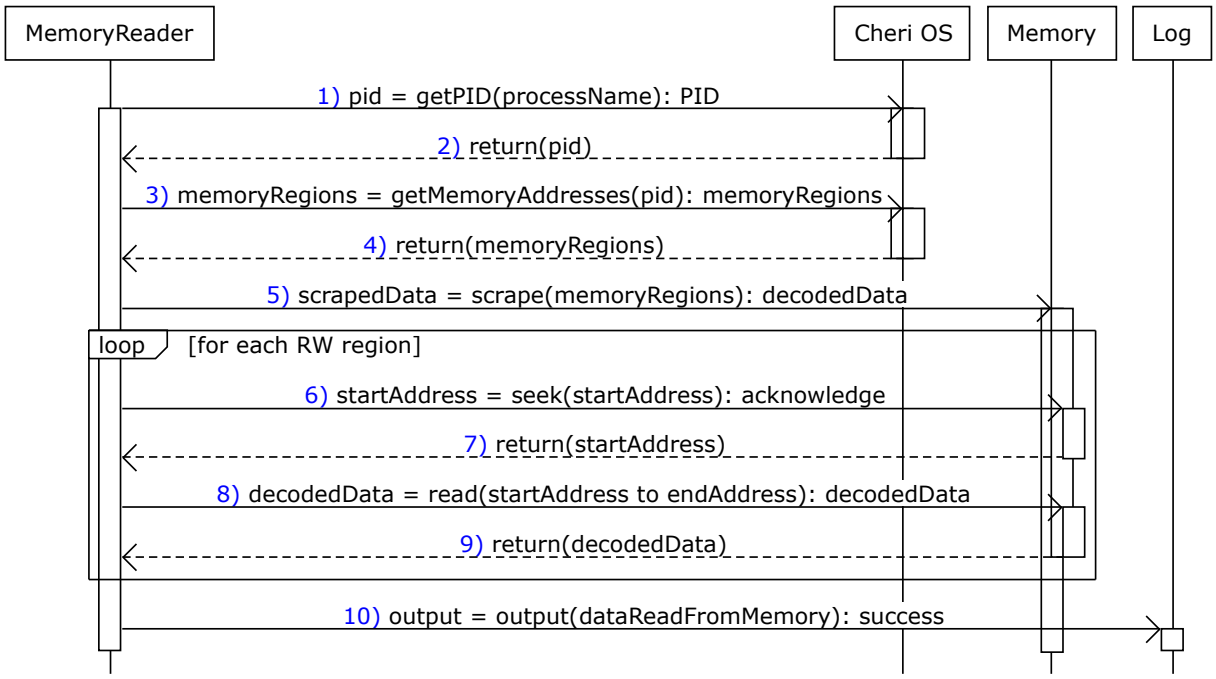


Figure 12. Procedure to scrap memory regions.

8.1. Results

Table 8 summarises the results. The columns have the following meaning:

[20] `https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-single-compartment-performance/memory_reader.py`

Table 8. Memory isolation in executions within and without compartments.

Trial num.	Execution env.	User privileges	Access	Sensitive Data Visible
1	in Compartment	Root	Granted	Yes
2	in Compartment	Ordinary user	Denied	No
3	out Compartment	Root	Granted	Yes
4	out Compartment	Ordinary user	Denied	No

Test num:
Unique identification number of the test.

Execution env.:
The execution environment where the application is executed, either within a compartments or no compartment.

User privileges:
The privileges granted to the user that executes the `memory_reader.py` script.

Access:
The response of cheriBSD to the `memory_reader.py` script's to access the memory region.

Sensitive Data Visible:
Visibility of the data retrieved from the memory region. Can the visible to the `memory_reader.py` script extract information from the data?

The results shown in Table 8 indicate that a user with root privileges has permission to access any memory region, including memory regions allocated to compartments. However, ordinary users are unable to access memory regions allocated to processes including processes not executed inside compartments.

These results indicate that the Morello Board implements the traditional asymmetric trust model where user applications trust privileged software. Some applications demand the symmetric trust model where privileged software and user applications distrust each other. Examples of technologies that implement mutual distrust are Intel SGX and AWS Nitro Enclaves.

Observations runs of the experiment We observed some unexpected behaviours and crashes of the cheriBSD that demanded reboot to recover. We have no sound explanations, we only suspect that these issues are related to the memory managements in the Morello Board.

- **Process terminated by the OS:** We have observed that the application was terminated (i.e. killed) automatically by the cheriBSD OS, approximately, after 1 hour of execution. See Fig. 13.
  This behaviour seems to be related to the CheriBSD system's resource management. It seems that the operating system terminates processes that are consuming excessive memory or CPU, possibly in response to an infinite loop or undesirable behaviour.
  Another speculation is that the CHERI security model abruptly terminates processes that systematically attempt to access protected memory regions, illegally.

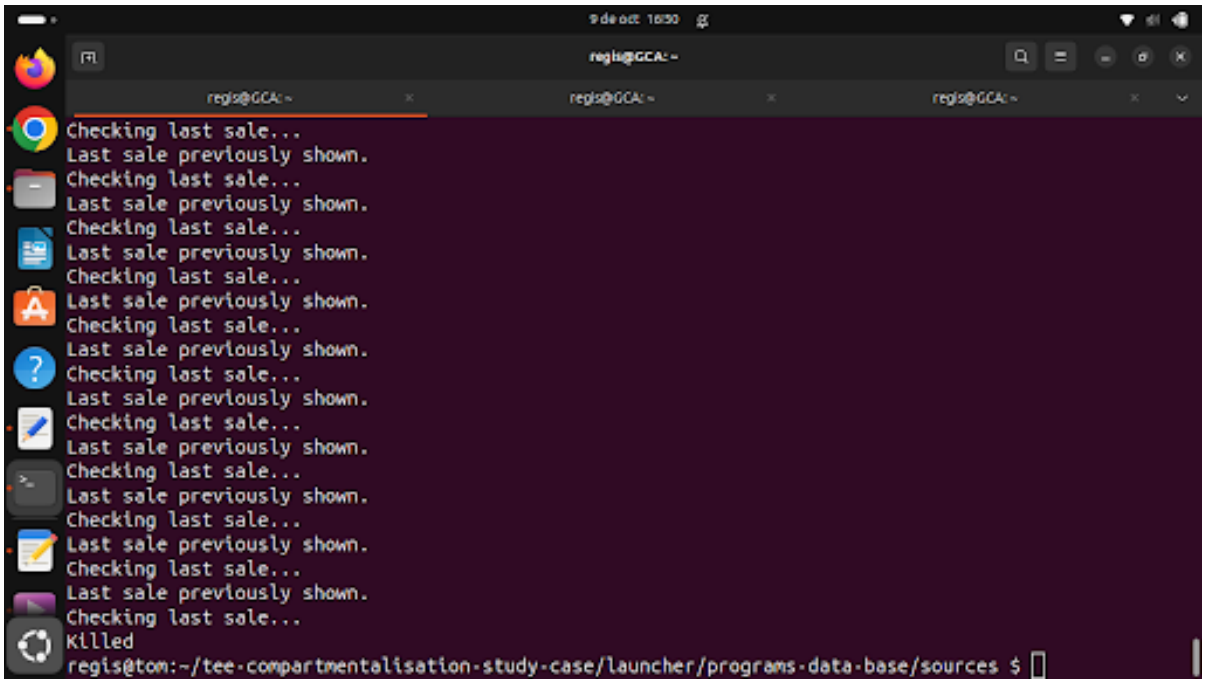


Figure 13. Abruptly termination of process by the OS.

- **Crash of cheriBSD OS**
  We have observed systematic crashes of the cheriBSD OS when the `memory_reader.py` script attempted to read a specific range of memory addresses.
  As shown in Fig. 14 the OS crashed reporting a `Broken pipe` error and the disconnection of the remote SSH shell when the `memory_reader.py` attempted to read addresses in the `0x4a300000` — `0x4bb00000` range. See Fig. 15.

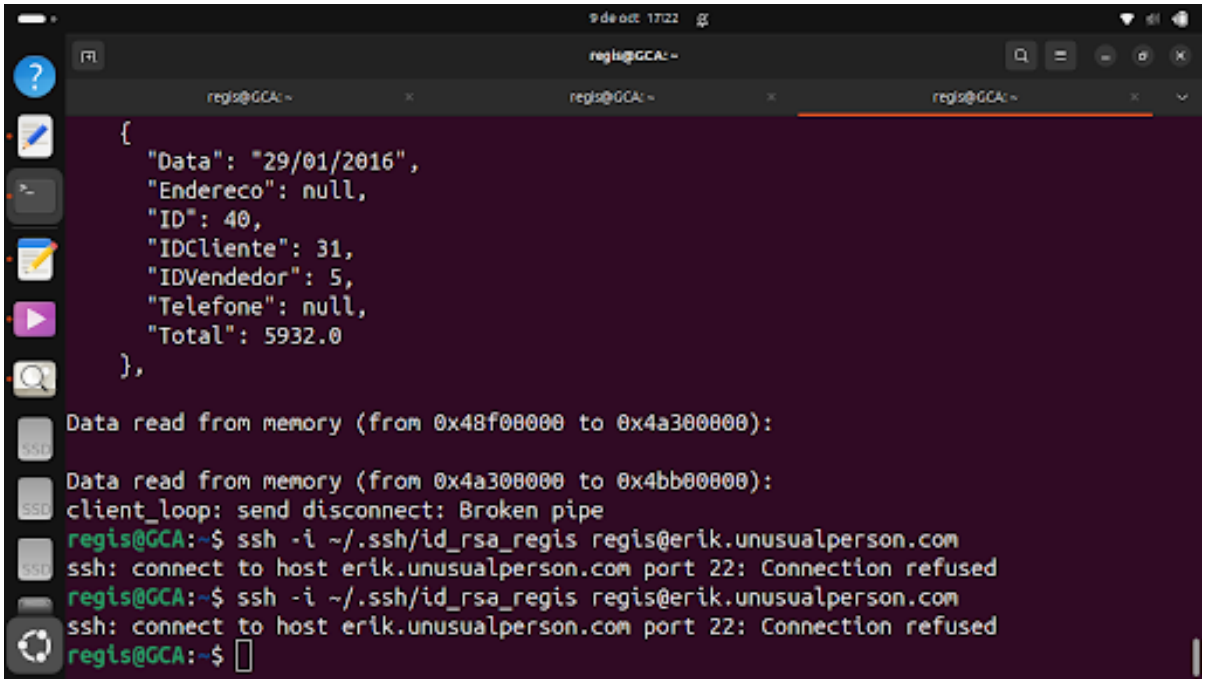


Figure 14. client_loop: send disconnect: Broken pipe.

```
3587          0x43000000          0x433e0000 -----    0    0   0   0 G----- gd
3587          0x433e0000          0x43400000 rw-RW    3    3   1   0 ---D-c sw
3587          0x43400000          0x43680000 rw-RW  583 101679  21   0 -----c sw
3587          0x43680000          0x43980000 rw-RW  764 101679  21   0 -----c sw
3587          0x43980000          0x43d00000 rw-RW  882 101679  21   0 -----c sw
3587          0x43d00000          0x44100000 rw-RW  997 101679  21   0 -----c sw
3587          0x44100000          0x44600000 rw-RW 1277 101679  21   0 -----c sw
3587          0x44600000          0x44c00000 rw-RW 1485 101679  21   0 -----c sw
3587          0x44c00000          0x45300000 rw-RW 1788 101679  21   0 -----c sw
3587          0x45300000          0x45b00000 rw-RW 2044 101679  21   0 -----c sw
3587          0x45b00000          0x46500000 rw-RW 2559 101679  21   0 -----c sw
3587          0x46500000          0x47100000 rw-RW 3065 101679  21   0 -----c sw
3587          0x47100000          0x47f00000 rw-RW 3584 101679  21   0 -----c sw
3587          0x47f00000          0x48f00000 rw-RW 4074 101679  21   0 -----c sw
3587          0x48f00000          0x4a300000 rw-RW 5114 101679  21   0 -----c sw
3587          0x4a300000          0x4bb00000 rw-RW 6120 101679  21   0 -----c sw
3587          0x4bb00000          0x4d700000 rw-RW 7163 101679  21   0 -----c sw
3587          0x4d700000          0x4f700000 rw-RW 8124 101679  21   0 -----c sw
3587          0x4f700000          0x51f00000 rw-RW 10225 101679  21   0 -----c sw
3587          0x51f00000          0x54f00000 rw-RW 12119 101679  21   0 -----c sw
3587          0x54f00000          0x58700000 rw-RW 14025 101679  21   0 -----c sw
3587          0x58700000          0x5c700000 rw-RW 14359 101679  21   0 -----c sw
3587          0x5c700000          0x61700000 rw-RW 1328 101679  21   0 -----c sw
3587     0xfbfdbffff000      0xfbfdc0000000 rw---    1    0   1   0 C----c sw
3587     0xfbfdc0000000      0xfe0000000000 rw---  848    0   1   0 C----- sw
3587     0xffffbfeff000      0xffffbff80000 rw-RW    1    1   1   0 CN---c sw
3587     0xffffbff80000      0xfffffff60000 -----    0    0   0   0 G----- gd
3587     0xfffffff60000      0xfffffff80000 rw-RW    3    3   1   0 C--D-c sw
3587     0xffffffffe000    0x1000000000000 r-x--    1    1  44   0 ------ ph
```

Figure 15. Crashing memory range.

A possible explanation is that the crash is caused by illegal attempts to read memory addresses storing privileged software.
This crash raises concerns about a possible failure in memory isolation when accessed by processes, such as the `memory_reader.py` script. Another possibility is that the privileged software running in this memory range is particularly sensitive to illegal read attempts, causing cheriOS crashes. Further investigation is required to determine the exact causes.

- **Error after rebooting the cheriBSD OS**
  Attempt to read memory after rebooting to recover from a crash, outputs `[Errno 2] No such file or directory: '/proc/PID/mem'` (see Fig. 16). The error indicates that file `/proc/{pid}/mem`, which is used by `memory_reader.py`, is unavailable.
  script attempts to access to read a process's memory, was unavailable.
  After rebooting the Morello Board, the error `[Errno 2] No such file or directory: '/proc/PID/mem'` was recorded when trying to access the process memory, as shown in Fig. 16. This error indicates that the file `/proc/{pid}/mem`, which the script attempts to access to read a process's memory, was unavailable.

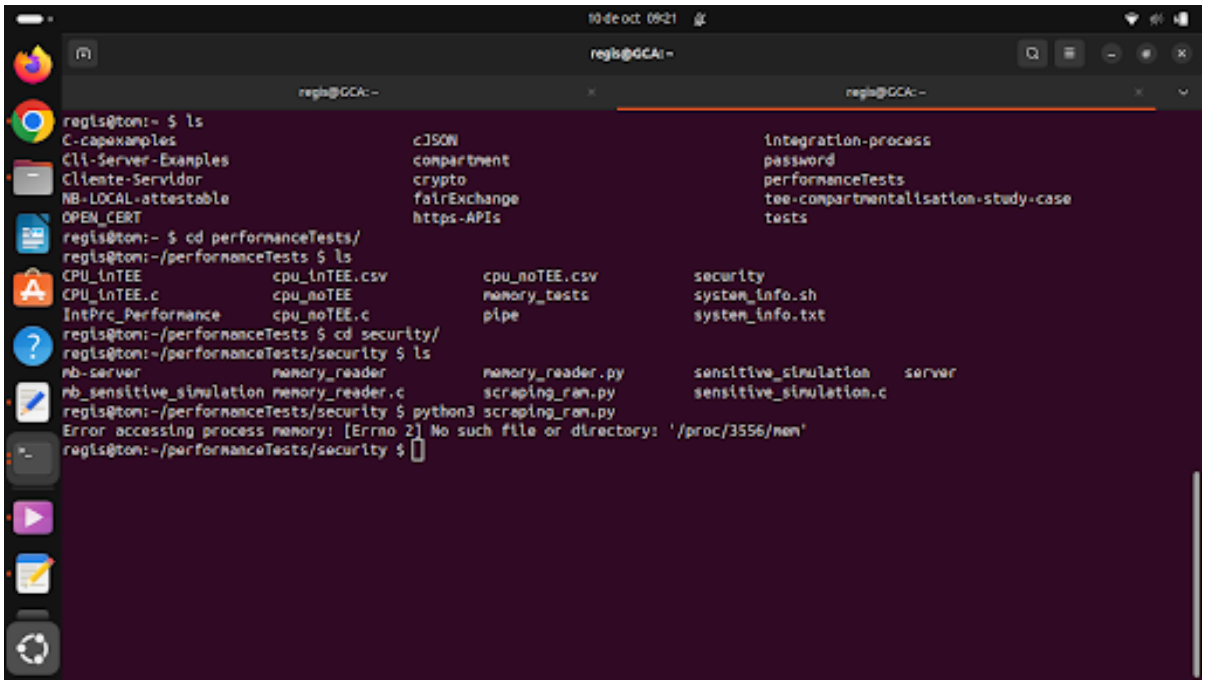


Figure 16. Error after recovering from a crash: [Errno 2] No such file or directory: '/proc/3587/mem'.

- **Procedure for running `memory_reader.py` after rebooting:**
  After rebooting to recover from a crash, it is necessary to verify that the `/proc` file system is mounted correctly mounted, the `mount` command can be used.
  ```
  $ mount | fgrep proc
  ```
  The following command can be used to mount `/proc` if it is not mounted.
  ```
  $ mount -t procfs proc /proc
  ```
  Once `proc` is mounted, the `memory_reaxder.py` script `memory_reader.py` script can be executed again.
  We believe that this behaviour is related to the persistence of cheriBSD configurations after rebooting from crashes. It might be useful to examine how resources are locked and released by cheriBSD after crashes.

9. Examination of memory isolation in executions with shared libraries

To explore memory isolation further, we executed a C program that communicates a parent and a child process over a pipe after compiling them using dynamic libraries. In this experiment we have the following C codes:

- `library_a.c`: the parent process that writes a string to one end of the pipe.
- `library_b.c`: the child process that reads the string from the other end of the pipe.
- `pipe-trampoline-in-experiment.c`: the main C program that creates the parent and child process when it is executed within compartments.

The compilation process is divided into two steps: Firstly, each individual module is compiled separately to create a dynamic library. Secondly, the main executable is compiled taking the dynamic libraries as input to create the main executable. In this example, we used two modules and therefore, we produce two dynamic libraries.

1. **Compilation of the parent library**:
   To create the object file `library_a.o` from the source file `library_a.c` execute:

   ```
   $ clang-morello -march=morello+c64 -mabi=purecap -fPIC -c
      library_a.c -o library_a.o
   ```

   The CHERI-specific settings used enables position-independent code (`-fPIC`), which is needed for creating dynamic libraries.
   To create the dynamic library `liblibrary_a.so` from the object file `library_a.o` execute:

   ```
   $ clang-morello -march=morello+c64 -mabi=purecap -shared -
      o liblibrary_a.so library_a.o
   ```

   The source C file is available from Git: `library_a.c`[21].
2. **Compilation of the child library**: The procedure to produce the library of the child process is similar.
   To create the object file `library_b.o` from `library_b.c` execute:

   ```
   $ clang-morello -march=morello+c64 -mabi=purecap -fPIC -c
      library_b.c -o library_b.o
   ```

   To create the dynamic library `liblibrary_b.so` from the object file `library_b.o` execute:

   ```
   $ clang-morello -march=morello+c64 -mabi=purecap -shared -
      o liblibrary_b.so library_b.o
   ```

   The source file is available from `library_b.c`[22].
3. **Compilation of the main program:** The main program is compiled and linked with the dynamic libraries (`library_a.so` and `library_b.so`) created above, they are assumed to be located in the current directory specified as `-L.`.

   ```
   $ clang-morello -march=morello+c64 -mabi=purecap pipe-
      trampoline-in-experiment.c -L. -llibrary_a -llibrary_b
      -o pipe_trampoline
   ```

[21] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/library_a.c

[22] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/library_b.c

The source C file is available from Git at `pipe-trampoline-in-experiment.c`[23].

4. **Execution of the main program:** We executed the main program within a compartment.
    - We set the `LD_LIBRARY_PATH` to enable the program locate the shared libraries in the current directory.

      ```
      $ export LD_LIBRARY_PATH=.
      ```

    - To run `pipe_trampoline` within compartments we executed the following command:

      ```
      $ proccontrol -m cheric18n -s enable ./
         pipe_trampoline
      ```

9.1. Examination of memory isolation

We have performed the following steps to examine memory.

1. **Initiation the parent and child processes:** We started the `pipe_trampoline` to initiate the parent and the child process. The parent writes a string to one end of the pipe and the child process reads it from the other end.
2. **Memory reading:** We executed the `memory_reader.py` script available from `memory_reader.py` to attempt direct memory reads:

   ```
   $ python3 memory_reader.py
   ```

3. **Reading process:** We executed the `memory_reader.py` script. It iterates through each RW memory region associated with the PIDs of the parent and child processes, trying to read the data from each region defines by start and end addresses. We displayed the results on the screen (see Fig. 17).

9.2. Results

We have divided the results into three sections.

9.2.1. Data read from memory:

are available from `memory-reading-result.txt`[24] and show data read from memory.

The results indicate that, even when running in a multi-compartment environment, a user with root privileges is able to access data from memory. We were able to extract data, including messages and data blocks.

As an specific example, we can report that the cheriBSD crashed when we tried to access the region 0xfbfdbffff000 to 0xfbfdc0000000 which is marked with `rw---`, that is, it is a protected region.

We have stored some examples of data read in `memory-reading-result.txt`.

[23] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/pipe-trampoline-in-experiment.c

[24] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/memory-reading-result.txt

It is sensible to think that cheriBSD blocked access to the region marked with `rw---` permission. However, the crash of cheriBSD, as a reaction, is intriguing. Further investigation is needed to fully understand the interaction between these permissions and the security policies applied to react to attempt to bypass the permissions.

9.2.2. Memory regions:

Are available from `memory-regions-result.txt`[25] and show different memory regions marked with different access permissions.

Memory regions with `rw-RW` permissions allow read access without crashing the cheriBSD OS; in contrast, regions marked with `rw---` – grant read access only to the owner process. Attempts to access these regions from a different process result in crashes; Fig. 17 shows an example. The screenshot shows the content of the memory at crash time.

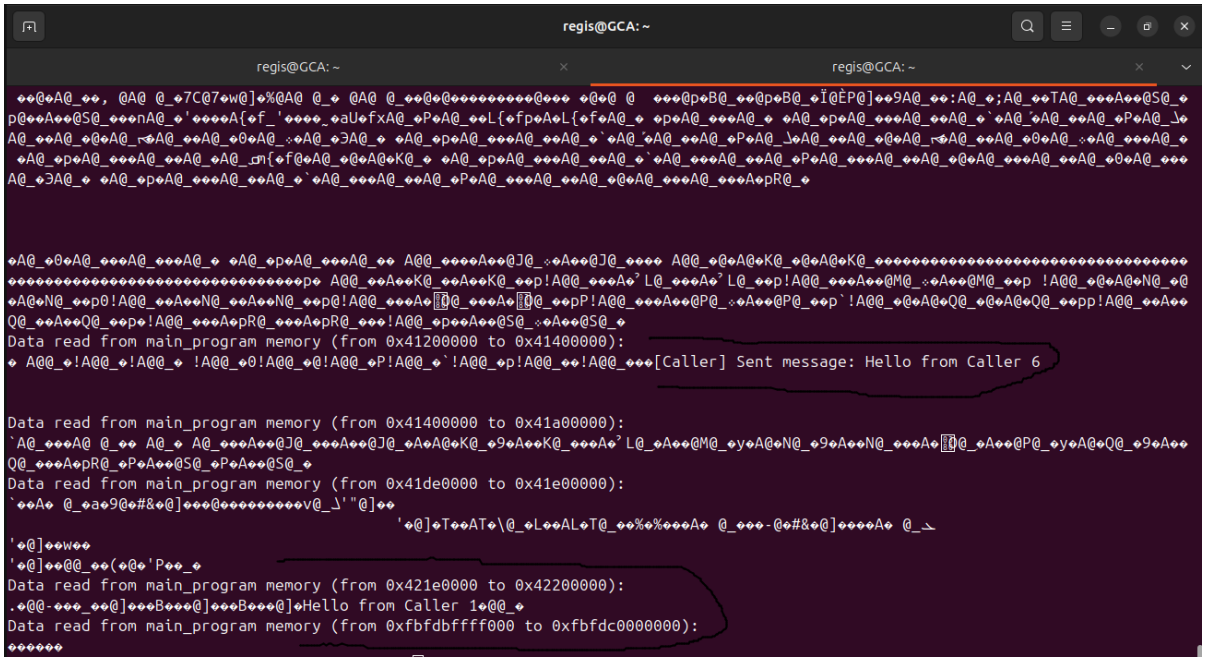


Figure 17. Memory read error: attempt to read region protected by compartments.

9.2.3. Execution results

Results are available from `execution-result.txt`[26] and show records of parent child communication over a pipe.

For example, line 205 (“msg received from child process TKYftt85v0l3d05SosZY1 ... iAbqS7D3VokIx”) shows the child process reading one of the strings with random characters sent by the parent process.

We managed to read this string directly from memory too. It is visible in the last lines of the raw version of the `memory-reading-result.txt` file.

[25] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/memory-regions-result.txt

[26] https://github.com/gca-research-group/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/execution-result.txt

10. Conclusions

This study evaluates the performance and security of library-based compartmentalisation on the Morello Board architecture using the cheriBSD 24.5 OS. The results indicate that CPU and memory operations within compartments are afflicted by moderate performance costs. The results reveal that the impact is visible in computationally intensive tasks such as complex mathematical operations (sin, cos, tan and exponentiation) and inter–process communication over pipes.

Regarding memory isolation, the results exhibit that users with root permissions are able to read memory areas allocated to compartments. The results only confirm that the current Morello Board implement an asymmetric trust model where privileged software is trusted by user applications.

References

[ARM 2019] ARM (2019). Morello program – arm®. https://www.arm.com/architecture/cpu/morello. [online: access in 10-jan-2024].

[Bartell et al. 2020] Bartell, S., Dietz, W., and Adve, V. S. (2020). Guided linking: dynamic linking without the costs. *Proc. ACM Program. Lang.*, 4(OOPSLA).

[Cheri Team 2022] Cheri Team (2022). Library based compartmentalisation. https://github.com/CTSRD-CHERI/cheripedia/wiki/Library-based-Compartmentalisation. visited on 28 Jan 2023.

[Cofta 2007] Cofta, P. (2007). *Trust, Complexity and Control: confodence in a convergent world*. Willey. https://www.zuj.edu.jo/download/trust-complexity-and-control-pdf-3/.

[Connolly 2024] Connolly, N. (2024). Using library compartmentalisation. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/cheritech24/02_02_Nick_Connolly.pdf. visited on 28 Oct 2024.

[DSbD programme 2024] DSbD programme (2024). Technology access programme. https://www.dsbd.tech/get-involved/technology-access-programme/. Visited on 20 Nov 2024.

[Gao 2024] Gao, D. (2024). Compartmentalization(7): Miscellaneous information manual compartmentalization(7). https://man.cheribsd.org/cgi-bin/man.cgi/c18n. visited on 28 Oct 2024.

[Gao and Watson 2024] Gao, D. and Watson, R. N. M. (2024). Library-based compartmentalisation on cheri. In *Proc. Programming Languages for Architecture Workshop (PLARCH'23)*. https://pldi23.sigplan.org/home/plarch-2023.

[Statistics How To 2023] Statistics How To (2023). Central limit theorem: Definition and examples. https://www.statisticshowto.com/probability-and-statistics/normal-distributions/central-limit-theorem-definition-examples/. [online: access in 23-oct-2024].

[Watson 2019a] Watson, R. (2019a). Capability hardware enhanced risc instructions (cheri). https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/. Visited on 20 Nov 2024.

[Watson 2019b] Watson, R. (2019b). Cheri software compartmentalization. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-compartmentalization.html. Visited on 29 Oct 2024.

[Watson and Davis 2024] Watson, R. N. M. and Davis, B. (2024). Getting started with cheribsd 24.05: Userlevel software compartmentalization (experimental). https://ctsrd-cheri.github.io/cheribsd-getting-started/features/c18n.html. Visited on 28 Oct 2024.

[Watson et al. 2020] Watson, R. N. M., Richardson, A., Davis, B., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N., Moore, S. W., Napierala, E., Sewell, P., and Neumann, P. G. (2020). Cheri c/c++ programming guide. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory.

[Watson et al. 2015] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. (2015). CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. IEEE Symposium on Security and Privacy*. https://ieeexplore.ieee.org/document/7163016.