Evaluation of performance and security strengths of library–based compartments created on Morello Boards winth cheriBSD 24.05: 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

2024-12-21

Abstract. *This report evaluates compartments created using the library–based compartmentalisation tool available on Morello Boards running the cheriBSD ver 24.05 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.

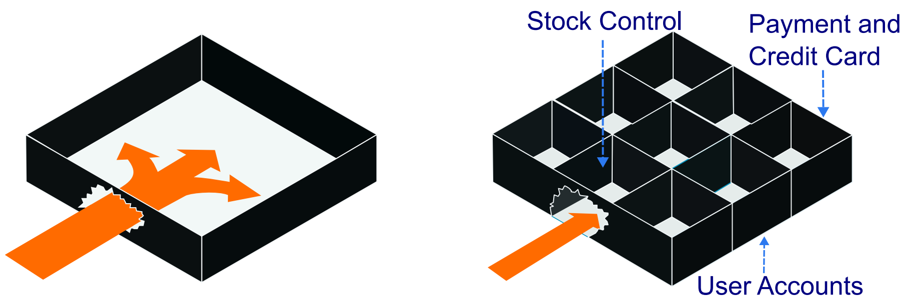
More technically, the idea to divide large complex software into components or modules and to run each component in isolation under the least privilege principle. As a result, the attacking surface and the impact of successful attacks is reduced.

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 re-organise 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 2024a]

- 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 2024a].

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. As of this writing (Dec 2024) Library–based compartmentalisation is an ongoing work on the Morello Board where they are being implemented on the basis of cheri-capabilities. Its architecture is described is a workshop paper [Gao and Watson 2024].

Transitions between domains (forwards and backwards) is controlled by a trampoline function generated and inserted by the dynamic linker. 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.

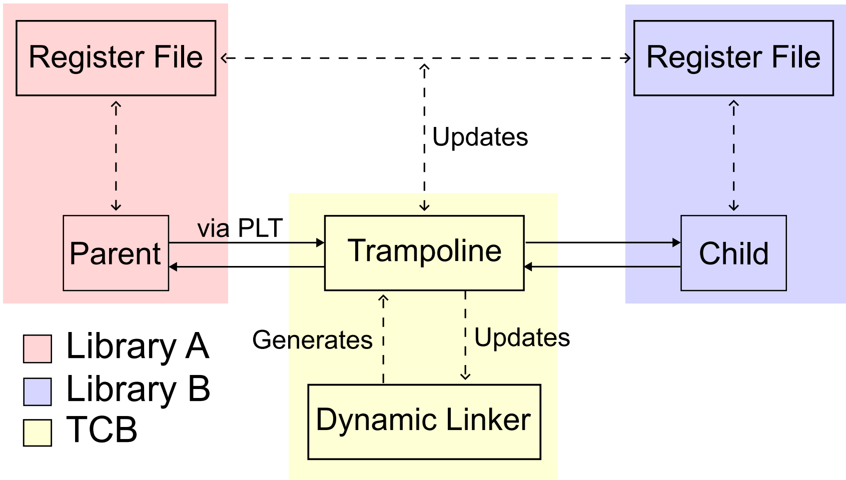


Figure 2. Dynamic linker generating a trampoline function between a parent and child process.

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.

2.1. Evaluation

Issues of concern are the performance cost that these compartments will incur and their security strengths. This report is meant to shed some light on these issues. 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. We have computed statistical estimations (means, media, etc.) and plotted results from metrics produced from 100 trails (we repeated the execution of each operation such as malloc) 100 times. The choice of 100 repetitions was based in on own experience in performance analysis.

3. Experiments set up

To run the experiments reported in this document, we use four Morello Boards connected as shown in Fig. 3.

- Three local Morello Boards are physically located in the William Gates building of the Computer Laboratory.
- A remote Morello Board physically located in Toronto, within the premises of TODAQ `https://engineering.todaq.net/`, a non-funding partner of the CAMB project `https://www.cl.cam.ac.uk/research/srg/projects/camb/`.

We connect to the remote Morello Board through ssh from a laptop connected to the network of the Applied Computing Research Group (GCA) `http://gca.unijui.edu.br/` at Unijuí, Brazil.

The figure shows the main configuration parameters of the Morello Board under evaluation. Table 1 lists additional parameters and the CheriBSD commands that can be used to double check the configuration parameters.

3.1. Compilation and execution

The inclusion of library-based compartments is determined at compilation and execution time. It is documented in:

- CHERI Software Compartmentalization [Watson 2019b], Robert Watson, 2019[1].
- Library based Compartmentalisation [Cheri Team 2022], Cheri team, 2022[2].
- Userlevel software compartmentalization (experimental) [Watson and Davis 2024b] Cheri team, 2024 [3].
- compartmentalization, c18n — library-based software compartmentalization [Gao 2024], Dapeng Gao, 2024[4].

[1] `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-compartmentalization.html`

[2] `https://github.com/CTSRD-CHERI/cheripedia/wiki/Library-based-Compartmentalisatio`

[3] `https://ctsrd-cheri.github.io/cheribsd-getting-started/features/c18n.html`

[4] `https://man.cheribsd.org/cgi-bin/man.cgi/c18n`

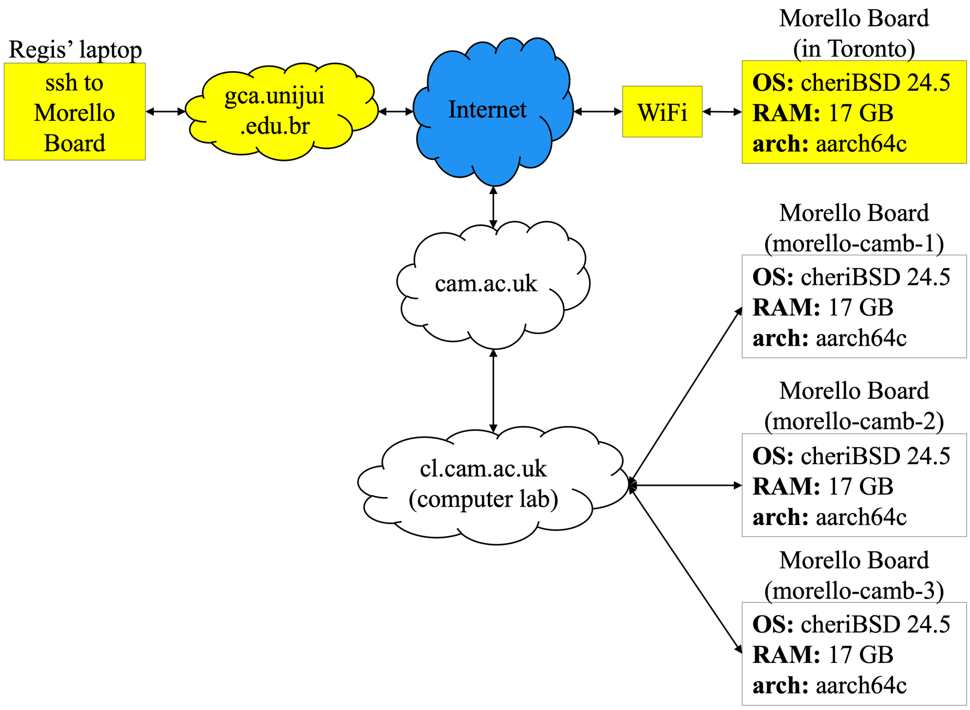


Figure 3. Morello boards used in the evaluation.

- Library-based Compartmentalisation on CHERI [Gao and Watson 2024], Dapeng Gao and Robert Watson, Plarch2023,[5].
- Dapeng's video-presentation [Gao 2023] of 2023, provides a summary of the architecture[6].

As explained in [Userlevel process environments](https://ctsrd-cheri.github.io/cheribsd-getting-started/features/processes.html), in CheriBSD 24.05, a user can compile his program to run in three different userspace execution environments:

- hybrid process: ??
- CheriABI processes: use -mabi=purecap
- Benchmark ABI processes: use -mabi=purecap-benchmark

The example of the compilation of helloworld.c `https://ctsrd-cheri.github.io/cheribsd-getting-started/helloworld/index.html`, might be helpful.

The `root# file` command can be used to verify the ABI targeted by the compiler.

Programs to be run in library-based compartments can be compiled either with `-mabi=purecap` or -`mabi=purecap-benchmark`. However, for performance evaluation, the latter alternative is recommended. See `man compartmentalization` `https://man.cheribsd.org/cgi-bin/man.cgi/c18n`. In our experiments, we have collected metrics from both alternatives for comparison.

[5]`https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/documents/LibraryBasedCompartmentalisationOnCHERI_Dapeng2023.pdf`

[6]`https://www.youtube.com/watch?v=0Zk0NzfiQJA`

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`

3.2. Compilation and Execution Without Library-based Compartments

The normal compilation (without the inclusion of library-based compartments) is demonstrated in the following example:

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

$ ./helloworld
```

3.3. Compilation and Execution with Library-Based Compartments

3.3.1. Compilation for purecap ABI

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

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

The meaning of the compilation parameters:

- `-march=morello+c64` parameter defines the 64-bit Morello architecture.
- `-mabi=purecap` flag targets the Application Binary Interface (ABI). With this ABI, all memory references and pointers are implemented as capabilities.

To execute the 'helloworld' program within a library-based compartment, the programmer can type:

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

The binary is executed within a library-based compartment that is enabled by the `proccontrol` command.

We follow example shown above in subsequent sections in the compilation and executeing of the programs used in the evaluation.

3.3.2. Compilation for purecap purecap-benchmark ABI

The compilation and execution with purecap-benchamark ABI are similar to the compilation and execution with purecap ABI, execept for the use of the `-mabi=puerecap-benchmark`.

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

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

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[7].

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 of this experiment is shown in Fig. 4.

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`[8]
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.

[7]Repository available at: `https://github.com/CAMB-DSbD/tee-compartimentalisation-study-case`

[8]`https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/max_num_of_compartments_performance/purecapABI_cheriOS_22.12/cheri-cap-experiment.py`

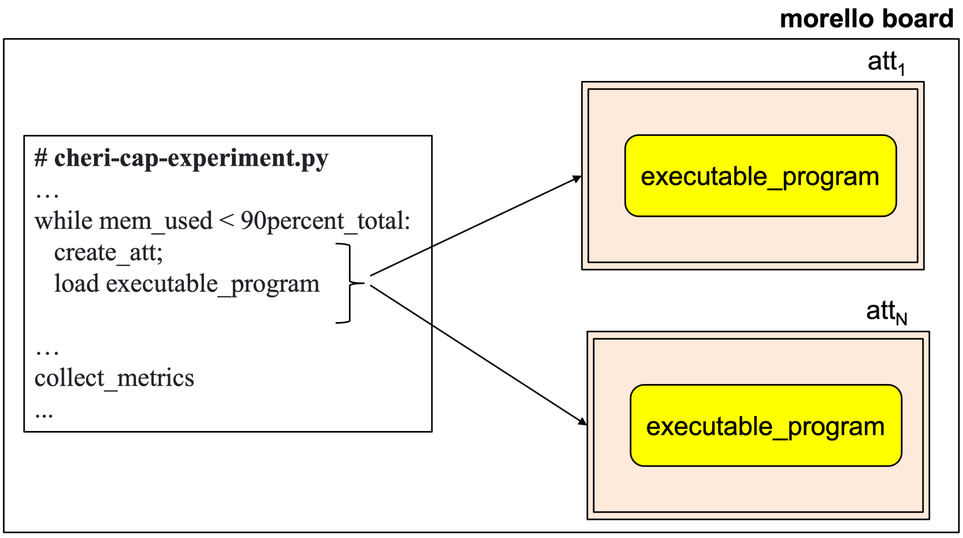


Figure 4. Max number of attestable that can be created before exhausting memory.

4.1. Results

We have stored the metrics collected in scv files. To help the reader, the first lines of the csv files are shown in tables to be read as follows:

Number of Compartments: The number of compartments created.
Memory Used (MB): The amount of memory consumed by the compartments.
Time Elapsed: The time elapsed from the start to completion of the experiment which is assumed to start at time zero.

We assume that the experiments start at time zero, with 0 number of compartments which have consumed zero MB of memory.

4.1.1. cheriOS ver 22.12

The `max_num_compart-experiment-cheriOS22.12-results.csv` file contains the metrics collected and is available from Git `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/max_num_of_compartments_performance/purecapABI_cheriOS_22.12/max_num_compart-experiment-cheriOS22.12-results.csv`.

Table 2 shows the first lines of the csv file.

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.

The blue line in the plot of Fig. 5 illustrates how memory is consumed as the number of compartments increases. The orange line indicates how many seconds takes to create a given number of compartments. For example, it takes 10 000 seconds to created 4 000 compartments.

We initially expected memory consumption to increase steadily from 1,628.3 MB, corresponding to a single compartment 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.

Table 2. Metrics of memory consumed by different numbers of compartments 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

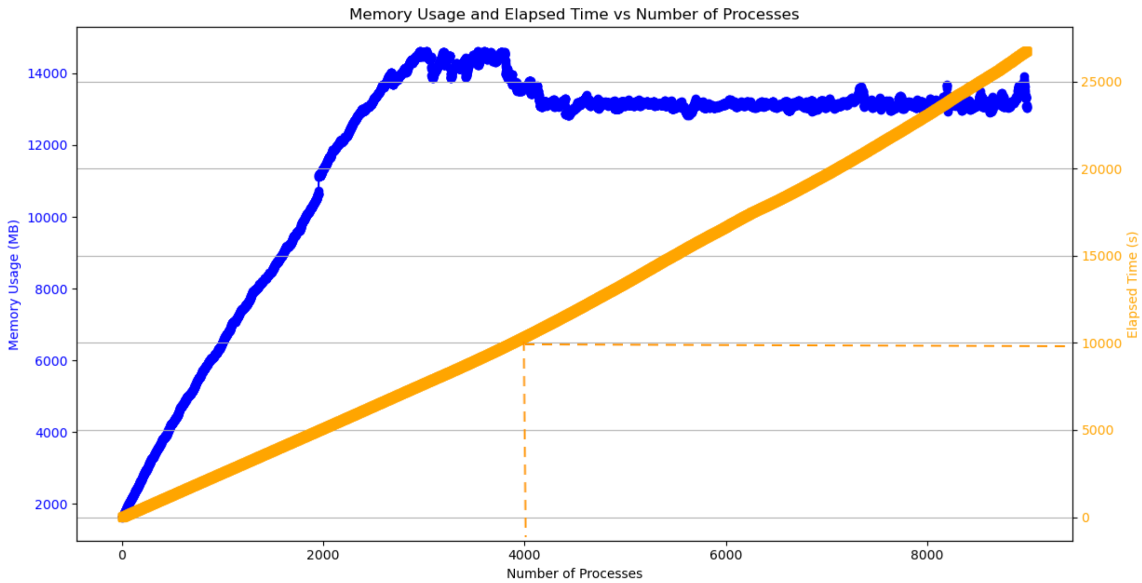


Figure 5. Max number of compartments created for purecap ABI created in cheriOS ver 22.12 and memory consumed.

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

We did not expect the behaviour 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 compartment replicas remains pending.

4.1.2. purecap ABI cheriOS ver 24.05

The `max_num_compart-experiment-purecapABI-results.csv` contains the results and is available from Git `https://github.com/`

CAMB-DSbD/tee-morello-performance-experiments/blob/main/max_num_of_compartments_performance/purecapABI/max_num_compart-experiment-purecapABI-results.csv.

Table 3 shows the first lines of the csv file.

Table 3. Metrics of memory consumed by different numbers of processes and elapsed time - purecapABI.

Number of Processes	Memory Used (MB)	Time Elapsed (ms)
1	1393.11	522.23
2	1399.96	1039.24
3	1404.45	1549.57
4	1411.93	2071.49
5	1421.18	2595.50
...	...	...
586	14728.59	300644.67

Fig. 6 shows a plot of the results collected in the csv file.

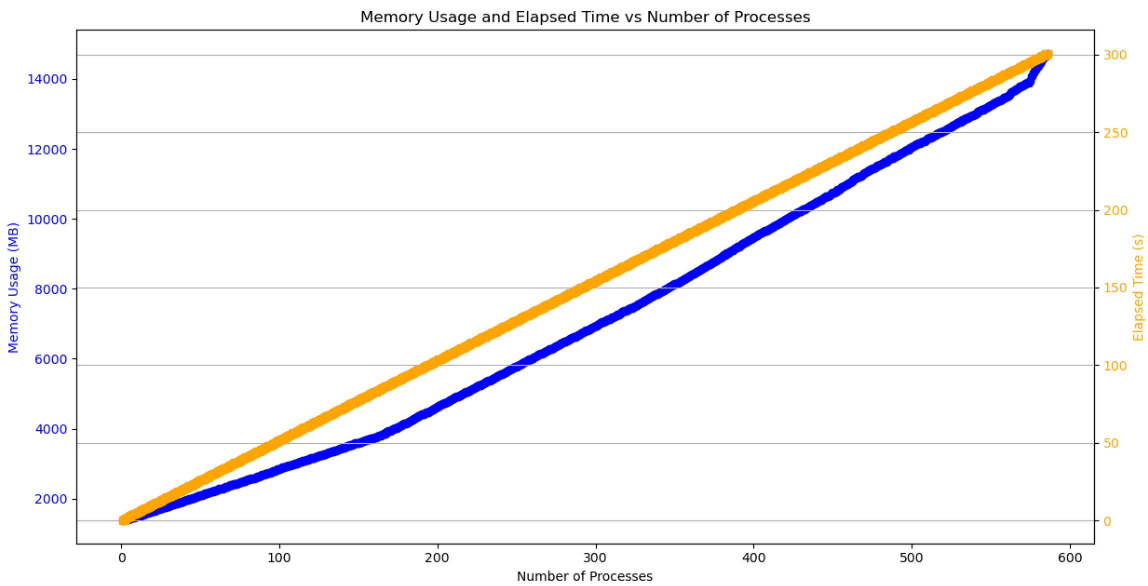


Figure 6. Max number of compartments created for purecap ABI created in cheriOS ver 24.05 and memory consumed.

4.1.3. purecap–benchmark ABI cheriOS ver 24.05

The max_num_compart-experiment-purecap-benchmarkABI-results.csv contains the results and is available from Git https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/max_num_of_compartments_performance/purecap-benchmarkABI/max_num_compart-experiment-purecap-benchmarkABI-results.csv.

Table 4 shows the first lines of the csv file.

Table 4. Metrics of memory consumed by different numbers of processes and elapsed time - purecap-benchmarkABI.

Number of Processes	Memory Used (MB)	Time Elapsed (ms)
1	1353.46	505.44
2	1357.30	1011.36
3	1360.80	1517.21
4	1364.79	2023.95
5	1368.99	2560.19
...	...	...
615	14691.81	315171.82

Fig. 7 a plot of the results collected in the csv file.

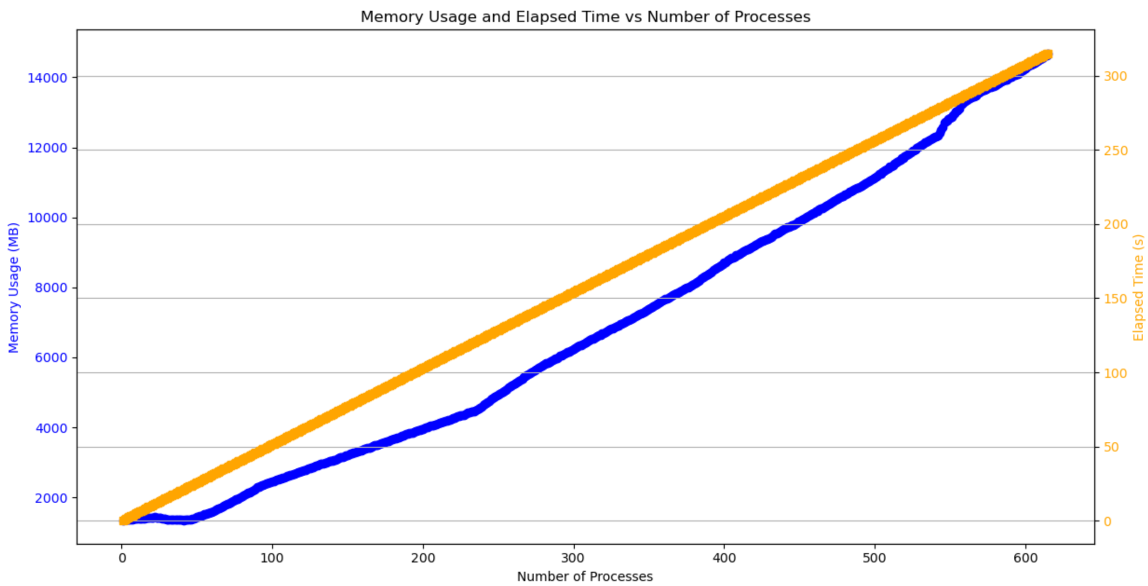


Figure 7. Max number of compartments created for purecap–benchmark ABI created in cheriOS ver 24.05 and memory consumed.

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

In this experiment, we use the code shown in Listing 1. It executes a list of operations on large blocks and measures the time as indicated on the right side.

- malloc: time taken to allocate the block of memory.
- write: time taken to write data to fill the entire memory block.
- read: time taken to read the data from the entire memory block.
- free: time taken to release the memory block.

As shown in Fig. 8 we use blocks of 100, 200, 300,...,100 000 MB. Blocks of these sizes are typical of applications that process images and access databases.

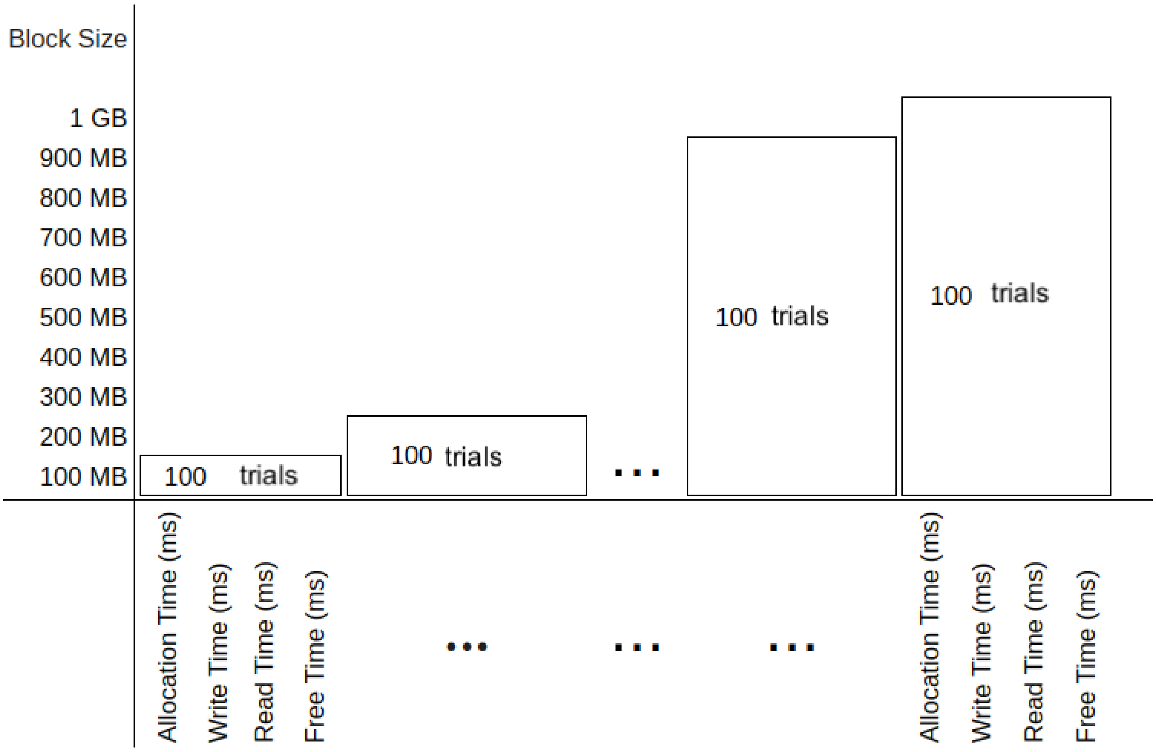


Figure 8. Performance of memory operations on memory blocks of different sizes

.

```
1  perform_tests(log_file, total_time)
2  begin
3   foreach block_size in MIN_BLOCK_SIZE to MAX_BLOCK_SIZE step
      BLOCK_STEP do
4     foreach test_num from 1 to num_of_trials do
5        allocation_time= time(malloc(block_size))
6        write_time= time(write_to_memory(block, block_size))
7        read_time= time(read_from_memory(block, block_size))
8        free_time= time(free(block))
9             log(log_file, block_size, test_num,
10       allocation_time, write_time, read_time, free_time)
11    endfor
12  endfor
13 end
```

Listing 1. Executes the memory operations measures their execution time and collects the metrics

The execution of the code shown in Listing 1 begins with the `perform_tests` function (line 1), which receives the name of an svc file as input to store performance metrics, including the total time taken to run the tests. 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 test `num_of_trials` times for each block size. `num_of_trials` is defined by the programmer either at compilation of run time.

At each iteration, the operations are executed, and their execution times are measured by the time function (lines 6–8): the time to write to the block is measured in line 6, the time to read the block is measured in line and, finally, the time to free the memory is measured in line 8. The metric collected are recorded in the log file along with the test number (line 9).

To collect metrics, we compiled C program shown in Algorithm 1 to be executed without compartments and within compartments:

5.1. Compilation and execution without compartments

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

$ ./memory-out-experiment
```

We stored the metrics collected in the memory-out-experiment-results.csv `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/memory-performance/outside-tee-execution/memory-out-experiment-results.csv` file. 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 Table 5

Table 5. Performance of memory operations executed 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

5.2. Compilation and execution withing compartments created to run in purecap ABI

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

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

The metrics collected are stored in two separate csv files: memory-in-experiment-purecap-results.csv `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/memory-performance/inside-tee-execution-purecap/`

`memory-in-experiment-purecap-results.csv` for the run inside a compartment. 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 Table 6.

Table 6. Performance of memory operations executed in compartments created to run in purecap ABI (looks odd!).

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

5.3. Compilation and execution withing compartments created to run in purecap-benchmark ABI

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

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

The metrics collected are stored in two separate CSV files: memory-in-experiment-purecap-benchmark-results.csv `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/memory-performance/inside-tee-execution-purecap-benchmark/memory-in-experiment-purecap-benchmark-results.csv` for the run inside a compartment. 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 Table 7.

5.4. Comparison of results

Plots of the results from Table 5 Table 6 and Table 7 are shown in Fig. 9.

- Allocation time (to update!): the results shown in Table 9 reveal that it takes longer to allocate memory blocks inside compartments. For example, the allocation of 100 MB takes 2 ms without a compartment, while it takes 106 ms inside a compartment. Allocation times vary from 1 to 3 ms without a compartment but from 106 to 265 ms inside a compartment. In contrast, the time to allocate memory within a compartment varies significantly from 106 to 265 and depends on the size of the block. Times range from 106 ms for 100 MB blocks to 251 ms for 700 MB blocks. In contrast, the time to allocate memory without compartments is shorter, it ranges from 2 to 7 ms for all block sizes.

Table 7. Performance of memory operations executed in compartments created to run in purecap-benchmark ABI (looks odd!).

Block Size (MB)	Allocation Time (ms)	Write Time (ms)	Read Time (ms)	Free Time (ms)
100	81 ± 158.99	40,369 ± 4.84	80,737 ± 7.56	86 ± 178.33
200	92 ± 219.79	80,737 ± 6.36	161,472 ± 10.22	210 ± 395.51
300	94 ± 295.34	121,105 ± 7.88	242,209 ± 12.70	219 ± 452.59
400	122 ± 430.07	161,472 ± 8.04	322,946 ± 17.29	425 ± 783.85
500	153 ± 596.27	201,842 ± 11.20	403,681 ± 14.85	215 ± 417.51
600	146 ± 646.07	242,210 ± 12.87	484,417 ± 17.45	436 ± 917.23
700	191 ± 879.02	282,579 ± 13.21	565,154 ± 18.71	453 ± 987.35
800	213 ± 1,088.59	322,947 ± 14.35	645,893 ± 17.43	822 ± 1,529.08
900	283 ± 1,535.56	363,315 ± 14.68	726,626 ± 17.13	818 ± 1,587.88
1000	246 ± 1,538.68	403,685 ± 15.61	807,368 ± 18.86	443 ± 1,004.74

- Write time (to update): Both tables show a linear increase in write time as the block size increases. However, execution inside a compartment takes longer. The difference becomes more evident when the sizes of the blocks increases.
- Read time (to update): The time to execute read operations increases linearly in both executions. However, execution within a compartment takes longer than execution without compartments.
- Free time (to update): 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 rages from 97 to 1 197 ms. In contrast, TableXX shows times that range from 3 to 9 ms in executions without compartments.

A boxplot is shown in Fig. 10.

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.

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.

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

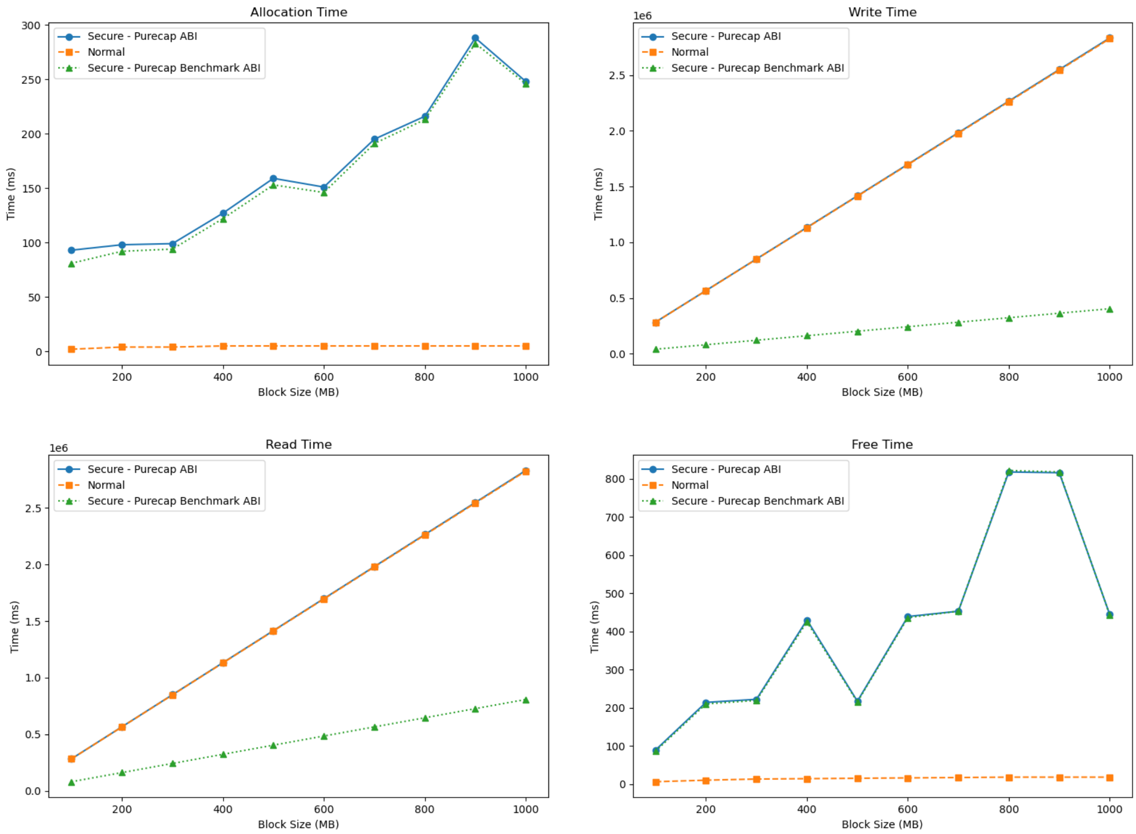


Figure 9. Comparison of time to execute allocate, write, read and release memory operations: no compartment, compartment for purecap and benchmark ABI.

```
1  perform_tests(log_file, total_time)
2  begin
3      for test_num in NUM_TESTS do
4          start_time = capture_time()
5            execute_operations(WORKLOAD_SIZE)
6          end_time = capture_time()
7          cpu_time =
8            calculate_cpu_time(start_time, end_time)
9            results(log_file, test_num, cpu_time)
10         total_time += cpu_time
11       endfor
12    end
```

Listing 2. Execute CPU demanding arithmetic operation and measure time taken to execute them

The execution begins with the perform_tests 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); this parameter is defined by the programmer either at compilation or runtime. Each iteration is a test identified

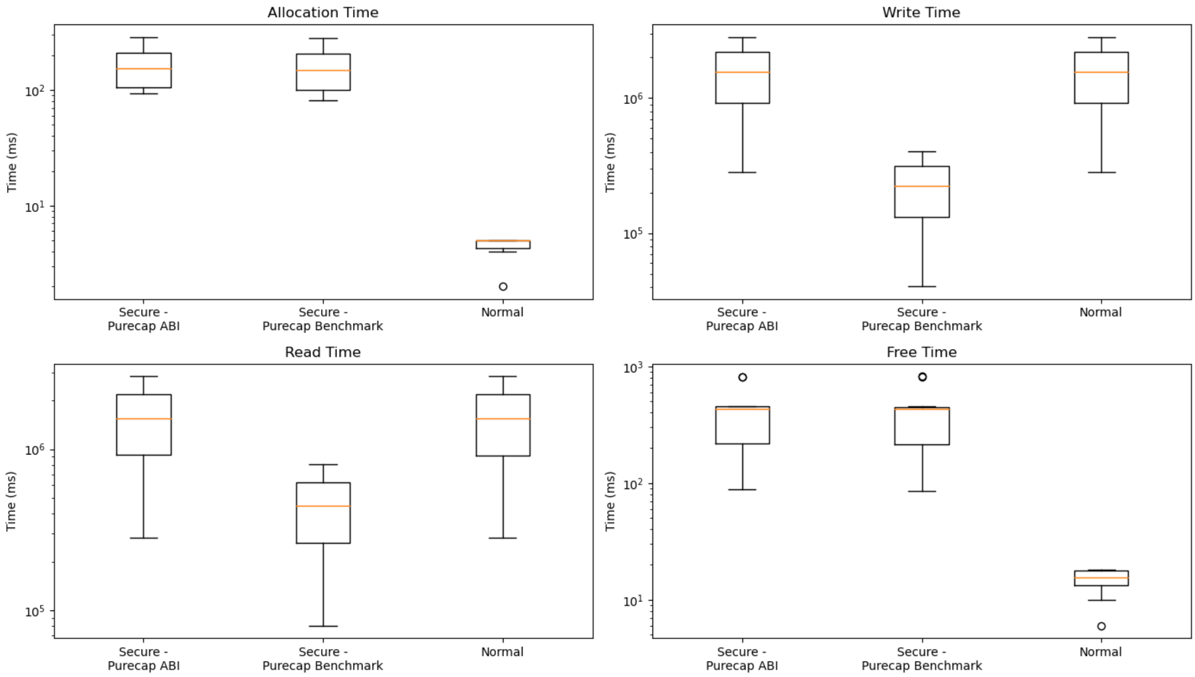


Figure 10. Comparison of dispersion of the time to execute allocate, write, read, and free operations: no compartment, compartment for purecap and benchmark ABI.

by `test_num`. In each iteration, the initial test time is recorded (line 4), followed by the execution of the 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 the test number (line 8), and added to `total_time`, that accumulates the total time spent on all the tests (line 9).

6.1. Compilation and execution without a compartment

We compile and run it as follows:

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

$ ./cpu-out-experiment
```

The source of the C program in available from cpu-out-experiment.c `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/cpu-performance/outside-tee-exection/cpu-out-experiment.c`.

The results collected from the execution are available from from cpu-out-experiment-results.csv `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/cpu-performance/outside-tee-exection/cpu-out-experiment-results.csv`.

6.2. Compilation and execution inside a compartment created for the purecap ABI

We compile and run it as follows:

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

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

The source of the C program in cpu-in-experiment-purecap.c `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/cpu-performance/inside-tee-execution-purecap/cpu-in-experiment-purecap.c`.

The results collected from the execution are store in cpu-in-experiment-purecap-results.csv `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/cpu-performance/inside-tee-execution-purecap/cpu-in-experiment-purecap-results.csv`.

6.3. 5.2 Compilation and execution inside a compartment created for the purecap-benchmark ABI

We compile and run it as follows:

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

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

The source of the C program is in cpu-in-experiment-purecap-benchmark.c `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/cpu-performance/inside-tee-execution-purecap-benchmark/cpu-in-experiment-purecap-benchmark.c`

The results collected from the execution are stored in cpu-in-experiment-purecap-benchmark-results.csv `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/cpu-performance/inside-tee-execution-purecap-benchmark/cpu-in-experiment-purecap-benchmark-results.csv`.

6.4. Comparison of results from the three experiments

Table 8 shows the time taken to execute CPU demanding arithmetic operations.

The results show that complex mathematical operations (trigonometric and exponential functions) executed within a compartment took 52,901 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 13.12%. However, the execution of arithmetic operations with integers without a compartment takes 922 ms, compared to 670 ms inside a compartment. The difference is a performance gain of 27.32%.

Similarly, the execution of floating point operations inside a compartment took 621 ms, which is lower than the execution without a compartment, which took 830 ms. This represents a performance gain of 25.18%. Finally, the execution of array manipulation operations took 101

Table 8. Times to execute CPU demanding arithmetic operations without a compartment and in compartments created for purecap and purecap-benchmark API.

Trial Type	CPU Time (ms) - Normal	CPU Time (ms) –purecap-benchmark	CPU Time (ms) - purecap
Maths (trigon. and exp. func)	46,759	52,901	70,780
Int	922	670	993
Float	830	621	804
Array	1,407	101	1,443

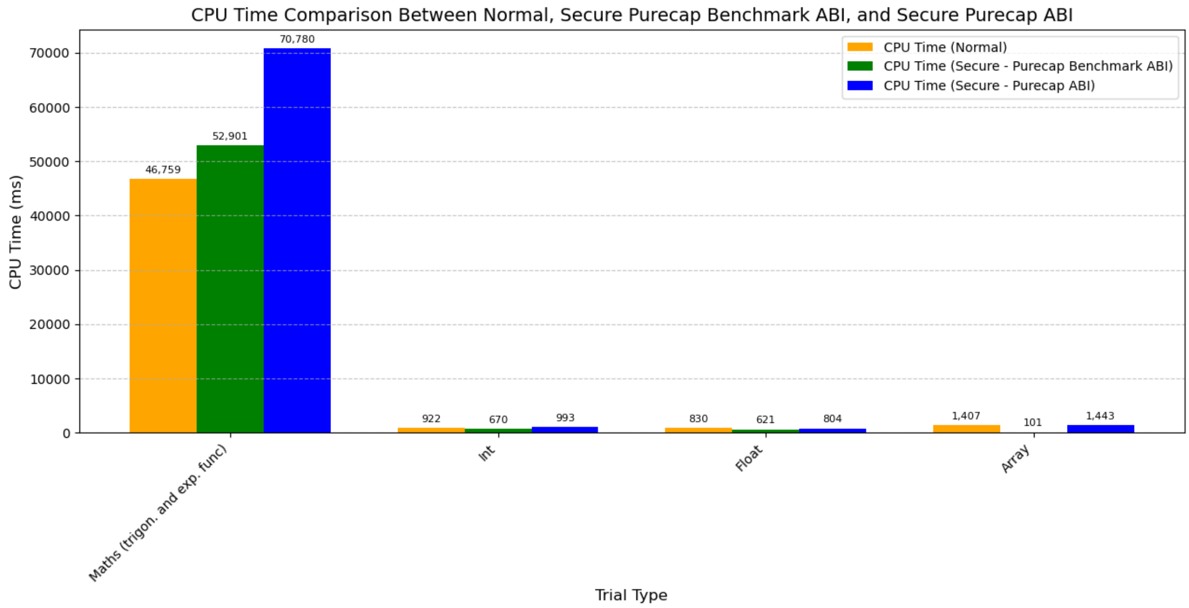


Figure 11. CPU performance in executions: no compartment, compartments created for purecap and purecap-benchmark.

ms inside a compartment, which is significantly lower than the 1,407 ms that it takes to execute the same operation without a compartment, representing a performance gain of 92.82%.

As visualized in Fig.11 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 significantly better inside compartments; strikingly, the float operations and array manipulation show substantial performance gains when executed inside a compartment.

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. 12, the parent process writes a message to the pipe and the child process reads it.

We run the C program within a compartments [9] and without compartments `pipe-out-experiment.c`[10]

- **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.

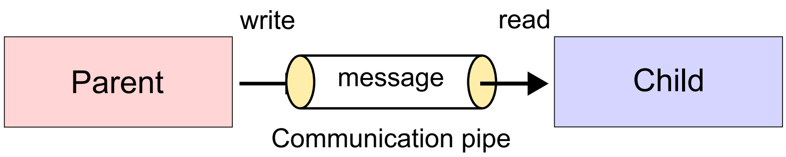


Figure 12. Parent–child communication over a pipe.

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].

Listing 3 codes the execution of the operations and the settings of timers to collect the metrics.

```
start_test(log_file)
 begin
   define STRLEN
   define NUM_OF_TRIALS
   for test_num from 1 to NUM_OF_TRIALS do
     if parent_process
        start_timer(write_time)
        write(pipe, message of size STRLEN)
        stop_timer(write_time)
```

[9] https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/pipe-performance/inside-tee-execution/pipe-in-experiment-result.c
[10] https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/pipe-performance/outside-tee-execution/pipe-out-experiment-result.c

```
10         write(pipe, write_time)
11       else
12         read(pipe, message of size STRLEN)
13         read(pipe, write_time)
14         start_timer(read_time)
15         stop_timer(read_time)
16         log(log_file, test_num, write_time, read_time)
17     endif
18   endfor
19 end
```

Listing 3. Pipe communication performance

Listing 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).

7.1. Results

We have stored the metrics from the execution of the C file in csv files. The files include five columns: the trial number (1 to 100), the size of the message sent over the pipe, the time it takes to read, the time it takes to read the message from the other en of the pipe and the total time taken to execute the read and write operations.

The metrics collected from the run of the experiment inside a compartment for the purecap ABI are stored in the pipe-in-experiment-purecap-results.csv () file.

Table 9 shows the first lines of the csv file.

Table 9. Time to execute write and read from a pipe inside a compartment created for the purecap ABI.

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 metrics collected from the execution of the experiment inside a compartment for the purecap-benchmark ABI are stored in the pipe-in-experiment-purecap-benchmark-results.csv () file.

Table 10 shows the first lines of the csv file.

Table 10. Time to execute write and read from a pipe inside a compartment created for the purecap-benchmark ABI.

Trial	Message Size (Bytes)	Write Time (ms)	Read Time (ms)	Total Time (ms)
1	1024	0.014	0.106	0.119
2	1024	0.001	0.001	0.003
3	1024	0.003	0.019	0.022
4	1024	0.003	0.024	0.027
...	...	...	...	...
100	1024	0.003	0.032	0.035

The file pipe-out-experiment-results.csv () stores the results of the experiment run without the use of a compartment.

Table 11 shows the first lines of the csv file.

Table 11. 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. 13.

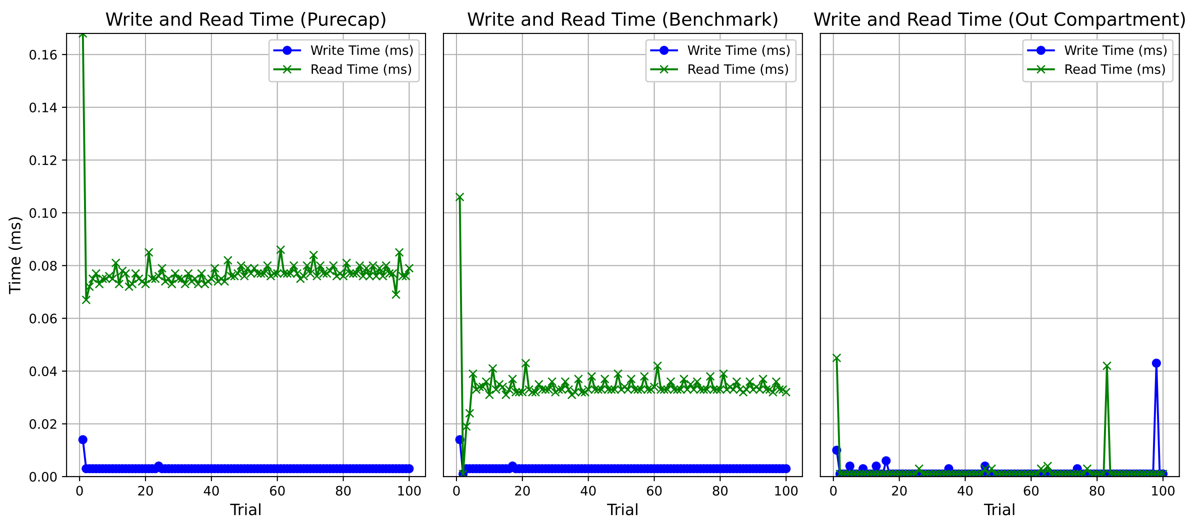


Figure 13. Times to write to and read a 1024 byte string from a pipe executed without using compartments and with the use of compartments created for purecap and purecap–benchmark compartments.

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. update

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`[11]
     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
```

[11] `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-single-compartment-performance/inside-tee-execution/integration_process-in-experiment.c`

- Compilation and execution without a compartment:
  The application integration is available from Git:
  `integration-process-out-experiment.c`[12]
  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
```

The `memory_reader.py`[13] script cycles through the memory regions of interest reading the data between the start and end addresses of each region directly.

Fig. 14 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.

8.1. Results

Table 12 summarises the results. The columns have the following meaning:

[12] `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-single-compartment-performance/outside-tee-execution/integration_process-out-experiment.c`

[13] `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-single-compartment-performance/memory_reader.py`

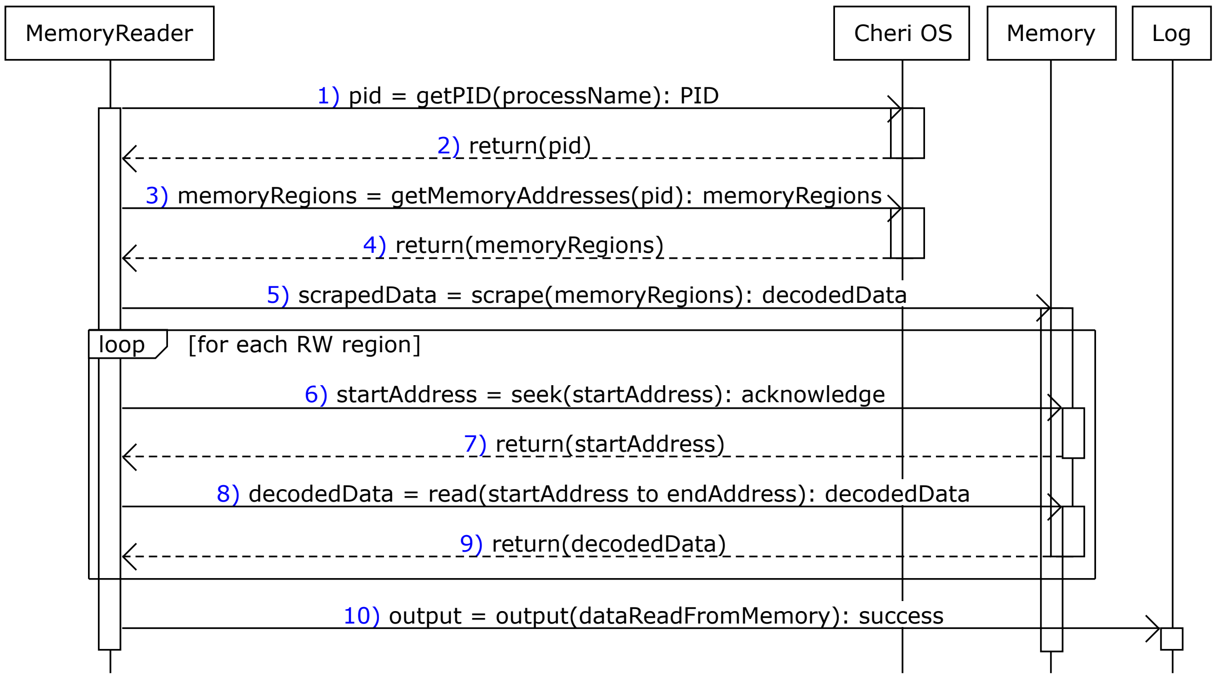


Figure 14. Procedure to scrap memory regions.

Table 12. 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 12 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.

8.2. Observations runs of the experiment

We observed some unexpected behaviour 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.

8.2.1. 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. 15.

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.

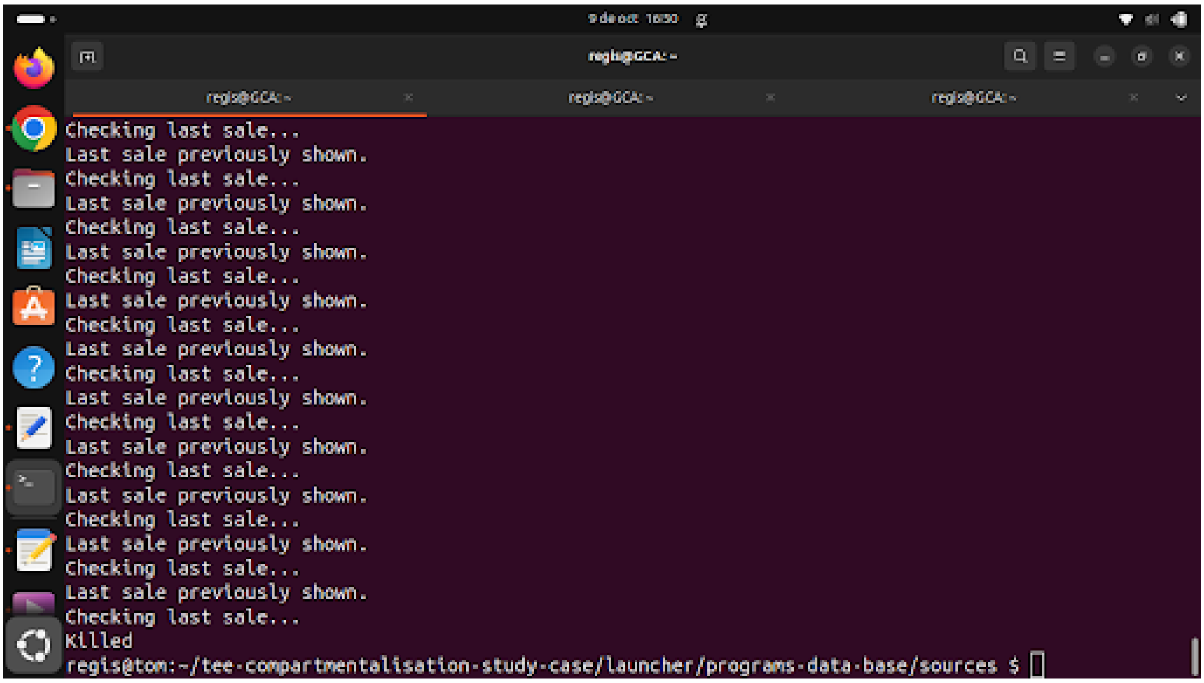


Figure 15. Abruptly termination of process by the OS.

8.2.2. 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. 16 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. 17.

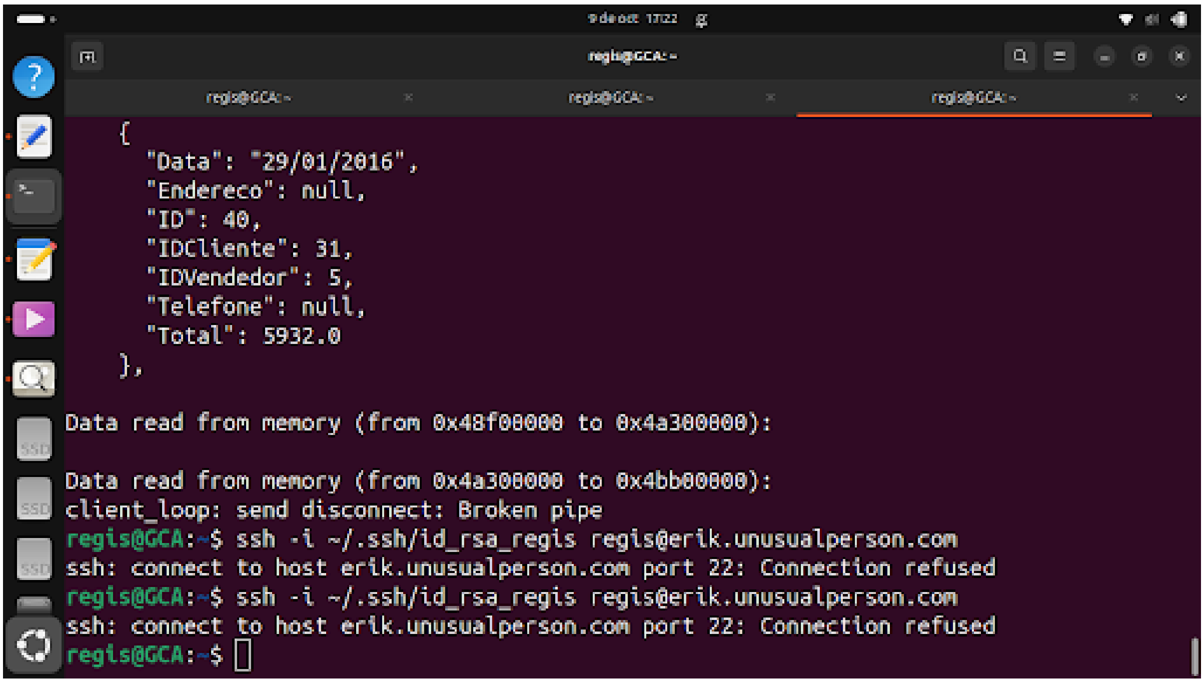


Figure 16. client_loop: send disconnect: Broken pipe.

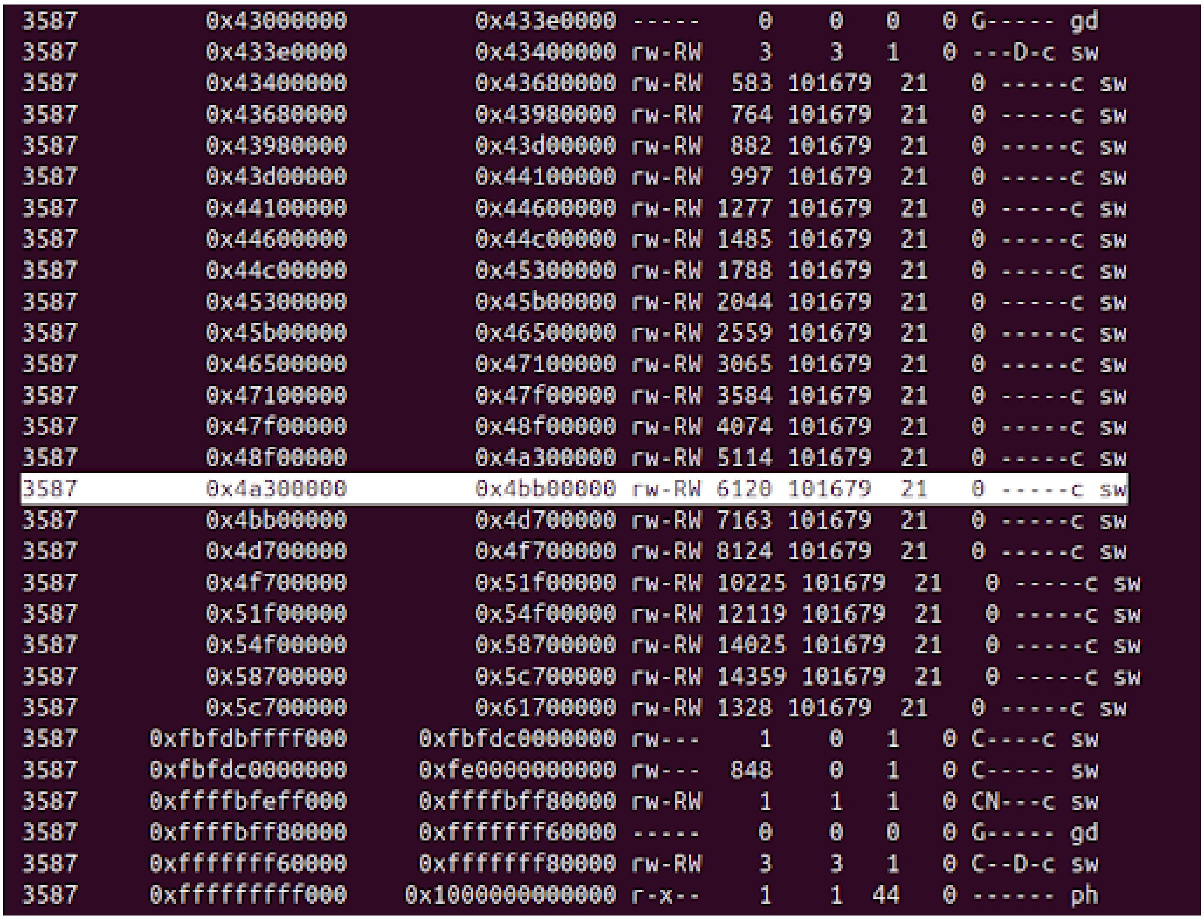


Figure 17. Crashing memory range.

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.

8.2.3. 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. 18). 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. 18. This error indicates that the file `/proc/{pid}/mem`, which the script attempts to access to read a process's memory, was unavailable.

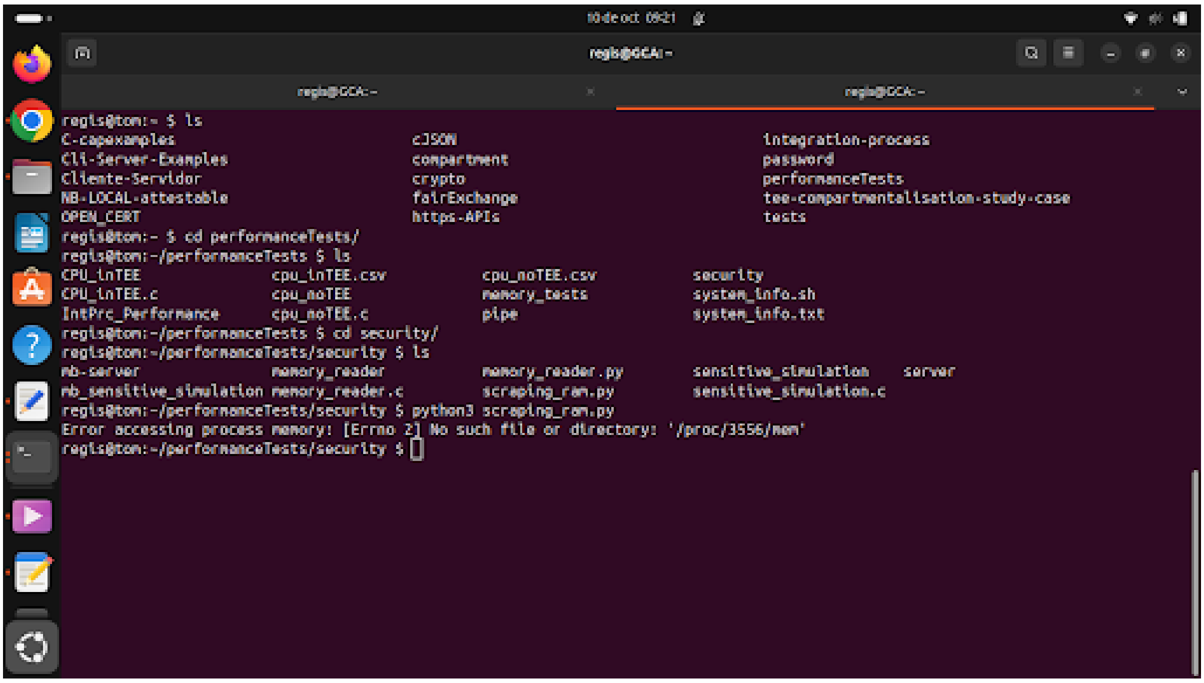


Figure 18. Error after recovering from a crash: [Errno 2] No such file or directory: '/proc/3587/mem'.

8.2.4. 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`[14].
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
   ```

[14] https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/library_a.c

The source file is available from library_b.c[15].

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
```

The source C file is available from Git at pipe-trampoline-in-experiment.c[16].

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. 19).

9.2. Results

We have divided the results into three sections.

9.2.1. Data read from memory:

are available from memory-reading-results.txt[17] and show data read from memory.

[15] https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/library_b.c
[16] https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/pipe-trampoline-in-experiment.c
[17] https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/memory-reading-results.txt

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-results.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-results.txt`[18] 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. 19 shows an example. The screenshot shows the content of the memory at crash time.

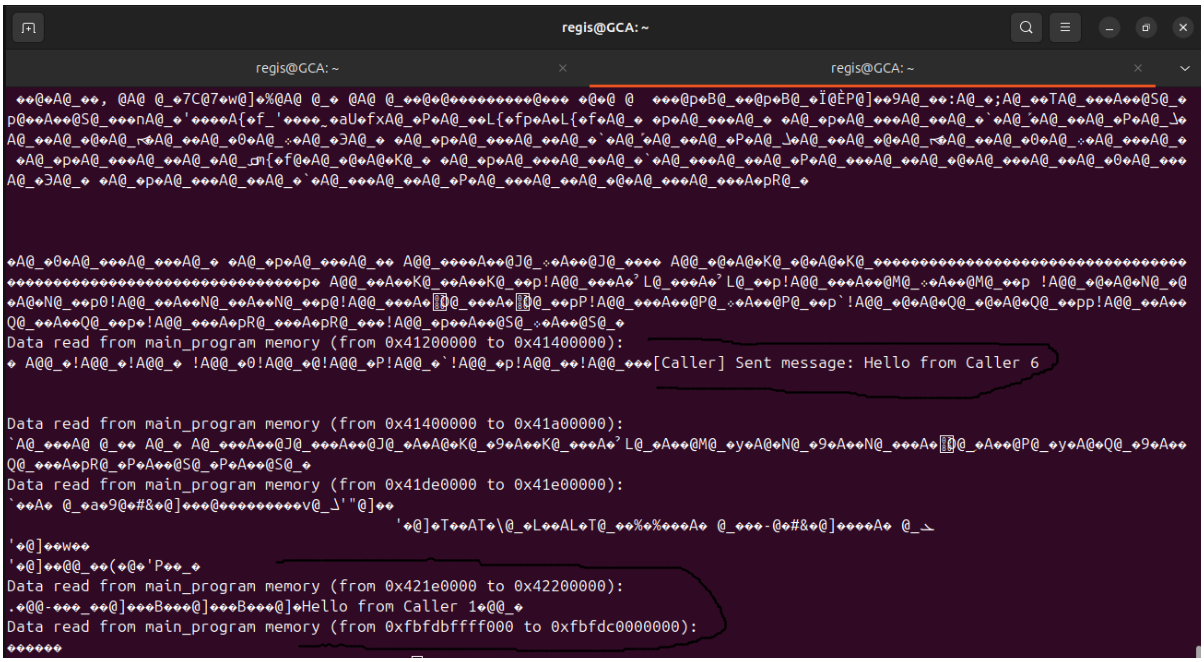


Figure 19. Memory read error: attempt to read region protected by compartments.

[18] `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/memory-regions-results.txt`

9.2.3. Execution results

Results are available from `execution-results.txt`[19] 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-results.txt` file.

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 2023] Gao, D. (2023). Library-based software compartmentalisation for cheri (video presentation). `https://www.youtube.com/watch?v=0Zk0NzfiQJA`.

[19] `https://github.com/CAMB-DSbD/tee-morello-performance-experiments/blob/main/security-multi-compartment-performance/execution-results.txt`

[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 2024a] Watson, R. N. M. and Davis, B. (2024a). 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 and Davis 2024b] Watson, R. N. M. and Davis, B. (2024b). Getting started with cheribsd 24.05: Userlevel software compartmentalization (experimental). `https://ctsrd-cheri.github.io/cheribsd-getting-started/features/c18n.html`. [online: access in 22-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`.