# Results of some performance and security evaluation against library-based compartments created in the Morello Board

Regis Schuch<sup>1</sup>, Rafael Z. Frantz<sup>1</sup>, Carlos Molina-Jiménez<sup>3</sup>

```
<sup>1</sup>Unijuí University - Ijuí - Brazil

<sup>2</sup>University of Cambridge - Cambridge - United Kingdom
{regis.schuch, rzfrantz}@unijui.edu.br,
carlos.molina@cl.cam.ac.uk
```

**Abstract.** This document presents the results of some performance and security evaluation tests conducted on library-based compartments created on a Morello Board running cheriBSD 24.5.

Attestable's performance evaluation In the CAMB project we are interested in using Morello Boards for creating attestables (compartments) to support the execution of exfiltration sensitive processes. The idea is to create an attestable to host the execution of the sensitive code of the exfiltration sensitive process. We account for attestables that remain active for several hours or days. The question that emerges here is how many compartments can created to run simultaneously on a single Morello Board before exahusting its RAM memory? Naturally, to a large extent the answer depends on the memory consumed by the code. In the experiments that we have conducted, we run a server that listen to connection

#### 1. Memory exhaustion by attestable replicas

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 loaded the attestable with a C program compiled with the library compartmentalisation tool and loaded in a compartment; precisely, we used the enterprise application integration (see yellow box) use case implemented in -tee-compartmentalisation-study-case repository

The metric 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.

Some experimental facts:

- (1) The Morello board used to conduct the experiment has 17,118,408,704 bytes (approximately 17,118.4 MB). Thus, 90 per cent of its memory is 15,406,567,833.6 bytes (approximately 15,406.5 MB).
- (2) In the experiments that we conducted, we loaded code of the EAI implemented in (see yellow box) tee-compartimentalisation-study-case repository We compiled as shown below:

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

(3) cheri-cap-experiment.py script is used to create the replicas of the attestables, and collect metrics. We incremented the number of replicas created from 1 to N. See replication of attestable results.

The Figure 1 shows the experiment set up:

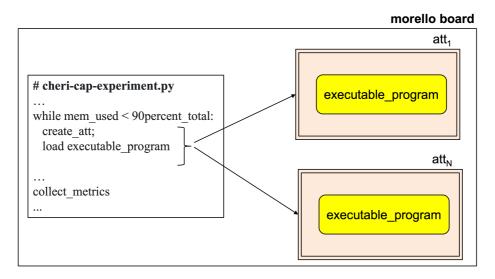


Figure 1. Memory exhaustion by attestable replicas.

The graph, shown in Figure 2 shows the behaviour of memory consumption and elapsed time as the number of testable replicas increases on the Morello board.

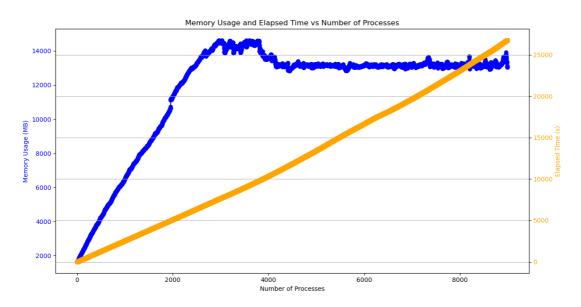


Figure 2. Memory consumption and elapsed time versus number of replicas on the Morello board.

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

- (1) **Initiation:** In the Morello board, Alice initiates *cheri-cap-experiment.py*.
- (2) **Launch:** To launch *cheri-cap-experiment.py*, Alice executes:

% python3 cheri-cap-experiment.py

(3) python3 cheri-cap-experiment.py runs incrementally, creating attestable replicas until it detects that the attestables have consumed 90% of the 17,118.4 MB of the Morello Board's memory, that is, about 15,406.5 MB.

#### **Preliminary observations:**

The results are shown in replication of attestable results and exhibit an unexpected behaviour. We expected memory consumption to increase steadily from 1,628.3 MB, which corresponds to a single attestable replica, to 15,406.5 MB (90% of total memory) consumed by N attestable replicas. The aim was to determine the value of N.

However, unexpectedly, memory consumption increased steadily only up to the creation of 3,800 attestable replicas, which consumed 14,582.5 MB. Beyond that, intriguingly, memory consumption decreases as the number of attestable replicas increases. Note that the last metric shows that 8,991 attestable replicas consume 13,066.4 MB (approximately 76% of the total memory).

We do not have an explanation for that yet. We need to double check measurements. We are showing this preliminary result only to show that this is a pending question in our agenda. The analysis of the time to wipe the memory of the attestable replicas is also pending.

The cpu-performance and memory-performance folders contain code and results of some experiments in progress. Cpu-performance is meant to measure the performance of code running inside attestables in the computation of some mathematical operations, integer arithmetic, floating point arithmetic, and array manipulation. Memory-performance is meant to assess the time it takes an attestable to allocate and free blocks of memory of different sizes.

#### 2. CPU Performance Tests on the Morello Board

The main aim of this experiment is to measure and analyse the performance of the Morello Board's CPU by comparing the results of identical tests carried out inside and outside a secure enclosure. The tests measured the time required to perform computational operations, including complex mathematical functions, arithmetic operations with integers, floating point operations and matrix manipulation. The complex mathematical functions included sine, cosine, tangent, square root and logarithm. The integer arithmetic tests focused on multiplication, division, subtraction and modulo operations. For floating point operations, the focus was also on multiplication, division and subtraction. The matrix manipulation test involved initialising matrices and performing operations such as multiplication and division. The execution time of each operation was measured and recorded. These tests were repeated 100 times for each operation, in both environments - inside the secure compartment and in the Morello Board's normal operating environment - and the results were saved in a CSV format file for each environment.

Algorithm 1 details how to carry out the CPU performance tests and record the results.

Execution begins with the perform\_tests function (line 1), which receives as a parameter a log file where the results will be stored and the total accumulated time needed to run the tests. The function enters a repeat loop that is repeated the number of times specified by NUM\_TESTS (line 3), where each iteration represents a test identified by test\_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 used is calculated by subtracting the start\_time from

## Algorithm 1 CPUPerformance

- 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 = calculate\_cpu\_time(start\_time, end\_time)
- 8: results(log\_file, test\_num, cpu\_time)
- 9: total\_time += cpu\_time
- 10: end for
- 11: end

the end\_time (line 7). This time is then recorded in the log file, along with the test number (line 8), and also added to total\_time, which accumulates the total time spent on all the tests (line 9).

The code used for the tests is available in the tee-morello-performance-experiments repository.

#### **Test procedure**

Suppose user Alice is conducting the experiment. To carry out the CPU performance tests and collect the results, Alice takes the following steps:

- 1. **Start:** Alice compiles and runs the test program in the two scenarios described above:
  - Inside the compartment: Alice runs the program cpu-in-experiment.c on the Morello Board, using the secure environment.
    - Compile: clang-morello -march=morello+c64 mabi=purecap -g -o cpu-in-experiment cpu-in-experiment.c -L. -lm
    - Run: proccontrol -m cheric18n -s enable ./cpu-inexperiment
  - Outside the compartment: Alice runs the program cpu-out-experiment.c in the Morello Board's normal operating environment.
    - Compile: clang-morello -o cpu-out-experiment cpuout-experiment.c -lm
    - Run: ./cpu-out-experiment
- 2. **Execution:** The program iterates automatically through the different types of operations, performing complex mathematical calculations, arithmetic operations with integers, floating point operations, and matrix manipulation.
- 3. **Repetition:** Each operation is repeated 30 times. The time of each operation is recorded in files in CSV format for both environments.

#### **Results**

The results of the tests carried out inside the secure compartment were stored in the file cpu-in-experiment-result.csv, while the results of the run in the Morello Board's normal environment were stored in cpu-out-experiment-result.csv. The Table [1] compares the average execution times for each type of operation in the two environments.

**Table 1. CPU Time Comparison** 

Test Type	CPU Time (ms) - Out Compartment	CPU Time (ms) - In Compartment
Maths	46,696	69,998
Int	923	993
Float	816	785
Array	1,419	1,460

# Analysing the results

According to the Table 1 the results show that, on average, complex mathematical operations within the secure compartment took 69,998 ms, while in the normal operating environment the time was 46,696 ms, representing a difference of approximately 49.74 per cent. For arithmetic operations with integers, the average time was 993 ms inside the secure compartment and 923 ms in the normal operating environment, a difference of 7.58 per cent. For floating point operations, the time inside the safe compartment was 785 ms compared to the normal operating environment (816 ms), showing a slight improvement of 3.80 per cent. As for matrix manipulation operations, the time inside the safe compartment was 1,460 ms, while in the normal operating environment it was 1,419 ms, indicating an increase of 2.89 per cent.

The Figure 3 illustrates the differences in performance between operations conducted within and outside the secure compartment.

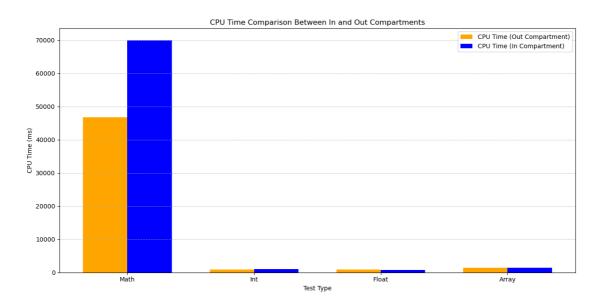


Figure 3. Comparison of CPU performance times in and out of the secure compartment on the Morello board.

These results suggest that the Morello Board's safe compartment introduces a small performance overhead in more complex operations, such as mathematical calculations. However, for simpler operations, such as floating point, the difference is minimal, with a slight increase in execution time within the secure compartment. This indicates that the safe compartment can be used efficiently in specific scenarios without significantly compromising performance, although the impact is more noticeable in operations that require more processing.

# 3. Memory Performance Tests on the Morello Board

The main aim of this experiment is to measure and analyse the performance of memory operations on a Morello Board by testing memory blocks of different sizes. Specifically, the test evaluates the time required to allocate, write, read and free memory blocks. To carry out this experiment, a C program was implemented - according to the formalism in Algorithm 2 - which automates the memory testing process. The programme runs repeated tests, increasing the size of the blocks at each stage and recording the times of each operation. The results are stored in a file in CSV format. The tests were carried out in two different environments: inside the TEE, where the programme was compiled for the CHERI architecture and run in a secure compartment of the Morello Board; and outside the TEE, where the programme was compiled and run in the normal operating environment of the Morello Board.

# Algorithm 2 MemoryPerformance

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

Algorithm 2 details how to perform memory performance tests and record the results. Execution begins with the perform\_tests function (line 1), which receives as parameters a log file where the results will be stored and the total accumulated time needed to run the tests. The function enters a repeat loop that iterates over different memory block sizes, defined from MIN\_BLOCK\_SIZE to MAX\_BLOCK\_SIZE with increments specified by BLOCK\_STEP (line 3). For each block size, another loop is started, repeating the tests the number of times defined by NUM\_TESTS (line 4). At each iteration, the memory allocation time is measured with the malloc function (line 5), followed by the time to write to the block (line 6), the time to read the block (line 7) and, finally, the time to free the memory with the free function (line 8). These times are recorded in the log file along with the test number (line 9). After all the iterations for all the memory blocks, the process is completed, allowing the performance of the memory allocation, write, read and free operations to be measured and compared in different block size configurations.

#### **Experimental Configuration**

1. The memory blocks tested range from 100 MB to 1 GB, with increments of 100 MB at each step. Each test involves 30 iterations per block size. The diagram in the Figure 4 shows the block pattern and the times for each operation associated with different block sizes.

The operations measured are:

a) **Memory allocation:** time required to allocate a block of memory.

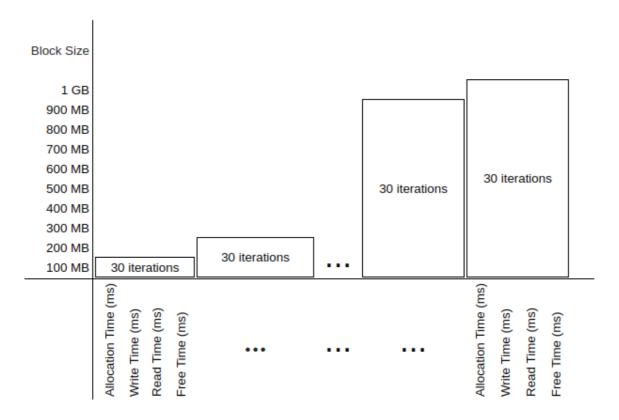


Figure 4. Memory performance at different block sizes on the Morello board.

- b) **Writing to Memory:** time required to write data to the entire allocated memory block.
- c) **Memory read:** time taken to read the data from the entire memory block.
- d) **Memory release:** time taken to release the memory block back into the system.
- 2. The code for the experiment we carried out is available in the tee-morello-performance-experiments repository.

## **Test procedure**

Imagine that user Alice is conducting the experiment. To carry out the memory performance tests and collect the results, Alice performs the following steps:

- 1. **Start:** Alice compiles and runs the test programme in two different scenarios:
  - Inside the compartment: Alice runs the programme memory-in-experiment-result.c on the Morello Board, using the secure environment.
    - Compile: clang-morello -march=morello+c64 -mabi =purecap -g -o memory-in-experiment memory-in-experiment.c -L. -lm
    - Run: proccontrol -m cheric18n -s enable ./memoryin-experiment
  - Outside the compartment: Alice runs the programme memory-out-experiment-result.c in the normal operating environment of the Morello Board.

- Compile: clang-morello -g -o memory-out-experiment
   memory-out-experiment.c -lm
- Run: ./memory-out-experiment
- 2. **Execution:** The programme automatically iterates over the memory block sizes, from 100 MB to 1 GB, performing the following sequence for each block size: allocate the block, write to the block, read the block, and release the block.
- 3. **Repetition:** For each block size, these steps are repeated 30 times, both inside and outside the compartment. The time of each operation is recorded and saved in a CSV file for the environment.

#### Results

For each CSV file, the iteration averages were calculated for each memory block size in the Allocation Time, Write Time, Read Time and Free Time attributes. For each memory block value (100 MB, 200 MB, etc.), the times measured in the iterations were added together and the average was obtained for each operation (allocation, write, read and free). This was done separately for the two files, representing in-compartment and out-of-compartment executions, respectively, as shown in the Tables 2 and 3.

**Table 2. In Compartment** 

Block Size (MB)	Allocation Time (ms)	Write Time (ms)	Read Time (ms)	Free Time (ms)
100	106	295.308	282.576	97
200	138	590.498	565.152	272
300	138	885.784	847.719	318
400	141	1,180.815	1,130.297	492
500	131	1,476.014	1,412.881	321
600	168	1,771.086	1,695.456	399
700	251	2,066.147	1,978.022	725
800	235	2,361.646	2,260.586	738
900	312	2,656.590	2,543.166	1,197
1000	265	2,951.487	2,825.741	405

#### **Analysing the results**

The results illustrated in the graphs in Figure 5 reveal some trends in relation to the four main operations: allocating, writing, reading and freeing memory.

- Allocation time: the values resulting from the tests in the secure compartment range from 106 ms for 100 MB blocks to 251 ms for 700 MB blocks, with some variations across block sizes. In the Morello Board's normal operating environment, the allocation time is considerably shorter, ranging from 5 ms (100 MB) to 7 ms (900 MB).
- Write time: in both environments, the times follow a linear behaviour as the block size increases. In the secure compartment, the values start at 295,308 ms for 100 MB blocks and reach 2,951,487 ms for 1 GB blocks. In the normal operating environment, the times are slightly longer, ranging from 491,512 ms (100 MB) to 4,903,282 ms (1 GB).
- **Read time:** the results also show a linear behaviour in both environments. Inside the secure compartment, the read time starts at 282,576 ms (100 MB) and increases up to

**Table 3. Out Compartment** 

Block S (MB)	Size	Allocation Time (ms)	Write Time (ms)	Read Time (ms)	Free Time (ms)
100		2	282.574	322.944	3
200		2	565.140	645.880	4
300		1	847.708	968.814	5
400		3	1,130.294	1,291.766	7
500		3	1,412.856	1,614.708	6
600		2	1,695.426	1,937.630	6
700		2	1,977.999	2,260.573	8
800		2	2,260.593	2,583.527	8
900		2	2,543.151	2,906.485	8
1000		2	2,825.742	3,229.433	9

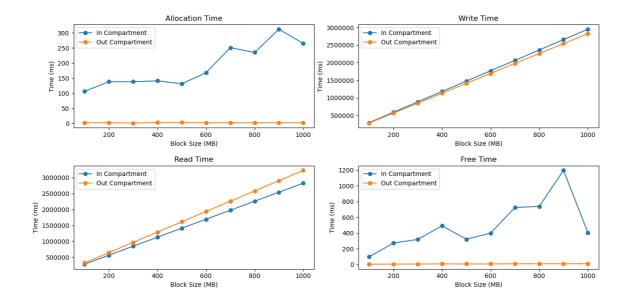


Figure 5. Performance analysis of memory operations on the Morello board.

2,825,741 ms (1 GB). In the normal operating environment, the times vary from 245,263 ms (100 MB) to 2,452,597 ms (1 GB), being slightly longer.

• **Memory release time:** shows similar behaviour in both environments. In the secure compartment, the time varies from 97 ms (100 MB) to 1,197 ms (900 MB), while in the normal operating environment the times are lower, ranging from 4,499 ms (100 MB) to 44,750 ms (1 GB).

Therefore, the results show that the secure environment imposes a greater burden on allocation time, with significant variations compared to the normal operating environment. On the other hand, write time and read time follow similar patterns, with the secure environment showing slightly higher results. However, the memory release time stands out, being significantly higher in the secure compartment. These results indicate that although the secure compartment imposes some overhead, it can still be used efficiently for memory operations.

# 4. Morello Board Pipe Communication Performance Tests

The main objective of this experiment is to measure the performance of communication between processes on the Morello Board using pipes, evaluating the time taken to write and read messages. The experiment was run both inside a secure compartment and outside the compartment, in the normal operating environment of the Morello Board. The aim is to compare the performance of these two scenarios and understand the impact of secure memory compartmentalisation on the communication time between processes.

To conduct this experiment, we developed a C programme that automates the exchange of messages between parent and child processes via a pipe, used here as a communication mechanism between processes. During the experiment, the child process is tasked with writing messages of a predetermined length to the pipe. At the same time, the parent process reads these messages, recording the write and read times. The programme has been specifically configured to test messages of a defined size, with each test timing the writing and reading times. The results of these tests are recorded in a file in CSV format. Algorithm 3 gives a detailed description of the experimental flow.

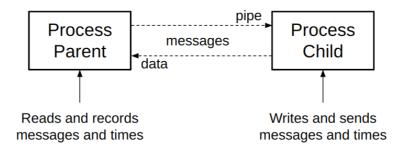
# Algorithm 3 Pipe Communication Performance

```
1: start_test(log_file)
2: begin
3: define STRLEN
4: define NUM OF MSG
5: for test_num from 1 to NUM_OF_MSG do
       if child_process then
6:
7:
           start_timer(write_time)
8:
           write(pipe, message of size STRLEN)
           stop_timer(write_time)
9:
           write(pipe, write_time)
10:
       else
11:
           read(pipe, message of size STRLEN)
12:
           read(pipe, write_time)
13:
           start_timer(read_time)
14.
15:
           stop_timer(read_time)
           log(log_file, test_num, write_time, read_time)
16:
       end if
18: end for
19: end
```

In Algorithm 3, the start\_test function (line 1) starts a sequence of operations involving writing and reading messages through a pipe. First, the STRLEN and NUM\_OF\_MSG parameters are set in lines 3 and 4, establishing the size of the messages and the total number of messages to be sent. For each test, which iterates from 1 to NUM\_OF\_MSG (line 5), the child process, if active, starts timing the write time (line 7), writes a message of size STRLEN to the pipe (line 8) and then records the write time (line 9), sending this time back to the parent process via the pipe (line 10). In parallel, the parent process reads the message and the write time from the pipe (lines 12 and 13), starts counting the read time as soon as it starts reading (line 14), and stops counting when it finishes reading (line 15). The write and read times, along with the test number, are recorded in the log file (line 16). This process is repeated until all the tests have been completed (line 17).

# **Experimental Configuration**

1. The following diagram (Figure 6) represents the experimental configuration used to evaluate communication between processes via a pipe. The data blocks, in this case represented by messages, vary by a standard size of 1024 bytes per message, and a total of 100 messages are tested to analyse the consistency and efficiency of communication between internal processes.



## **Configurations:**

- Message Size (STRLEN): 1024 bytes
- Number of Messages (NUM\_OF\_MSG): 100

Figure 6. Communication of the process via pipe.

The operations measured are:

- a) Writing to memory: time taken to write data to the pipe.
- b) **Memory read:** time taken to read the data sent via the pipe.
- 2. The code for the experiment we carried out is available in the tee-morello-performance-experiments repository.

# **Test procedure**

Imagine that user Alice is conducting the experiment. To carry out the pipe communication tests and collect the results, Alice performs the following steps:

- 1. **Start:** Alice compiles and runs the test programme in two different scenarios:
  - Inside the compartment: Alice runs the programme pipe-in-experiment.c on the Morello Board, using the secure environment.
    - Compile: clang-morello -march=morello+c64 -mabi =purecap -g -o pipe-in-experiment pipe-in-experiment.c -L.
    - Run: proccontrol -m cheric18n -s enable ./pipe-in -experiment
  - Outside the compartment: Alice runs the programme pipe-out-experiment.c in the normal operating environment of the Morello Board.
    - Compile: clang-morello -g -o pipe-out-experiment
      pipe-out-experiment.c
    - Run: ./pipe-out-experiment

- 2. **Execution:** The programme sends a total of 100 messages, each of a predefined size of 1024 bytes. For each message sent by the child process, the write time is measured and the message, along with the write time, is transmitted via the pipe to the parent process, which in turn records the read time as soon as the message is received.
- 3. **Repetition:** Each set of write and read operations is repeated 100 times for each environment inside the compartment and outside the compartment. The times of each operation are recorded in a file in CSV format for each environment.

#### **Results**

The results of the tests carried out inside the secure compartment were stored in the file pipe-in-experiment-result.csv, while the results of the normal ambient execution of the Morello Board were stored in pipe-out-experiment-result.csv. The Tables 4 and 5 shows the records for each data set:

**Table 4. Performance Data Inside the Compartment** 

Test	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

**Table 5. Performance Data Outside the Compartment** 

Test	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

The Tables 4 and 5 contain information on the test number, the size of the messages, the write and read times, and the total time for each operation.

#### **Analysing the results**

The graphs in the Figure 7 show the write and read times for each test in the two scenarios, inside and outside the secure environment (compartment) on the Morelo Board.

As we can see, the graphs reflect variations in write and read times, both inside and outside the safe room environment. There are fluctuations in the times recorded which may point to variability in the pipe's performance under different operating conditions. When comparing the graphs for the two environments, it is possible to notice differences in the times, which suggests that security configurations or differences in the operating environment can significantly influence communication efficiency.

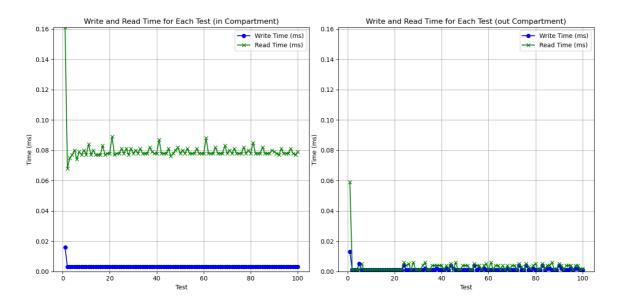


Figure 7. Comparison of write and read times for pipe communication in secure and normal environments on the Morello Board.

# 5. Testing the Exposure of Sensitive Data in Memory Regions through Direct Memory Access (Memory Scraping)

The main objective of this experiment is to measure and analyze the contents of the memory regions with read and write (RW) permissions of a target program running on the CheriBSD 24.05 operating system. This target program is the same one used in the Attestable's performance evaluation experiment, integration\_process. To do this, the experiment identifies the program's RW memory regions and attempts to directly extract the content stored in these regions, using the Python script memory\_reader.py, available in the tee-morello-performance-experiments repository. This experiment evaluates the potential for exposing sensitive data in memory by directly accessing areas where the program may store temporary or sensitive data.

# **Experimental Configuration**

The Figure 8 sequence diagram details how a programme's memory regions can be accessed directly from the operating system. To do this, the process implemented through the Memory Reader code asks the Cheri OS for the PID of the target process by name, getPID (processName), receiving the PID in response. Memory Reader then uses this PID to ask Cheri OS for a list of the memory regions that have read and write (RW) permissions associated with the process, getMemoryAddresses (PID), and the operating system returns the mapped regions. The Memory Reader then starts direct memory reading (scraping) by accessing the process's memory file and, for each RW region listed, fetches the region's starting address, reads the content between the starting and ending address, and receives the decoded data. This cycle is repeated for all RW regions, ensuring that all memory areas accessible for reading and writing are explored. Finally, the Memory Reader records the read data in the log, output (dataReadFromMemory).

#### **Test procedure**

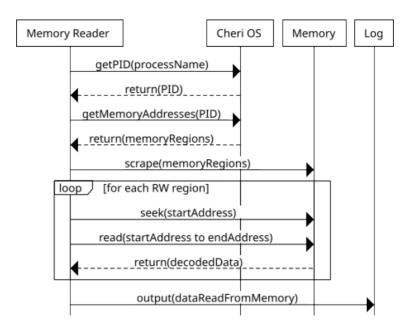


Figure 8. Sequence diagram detailing the memory scraping process through direct access in the Cheri OS.

To carry out the direct memory reading experiment, Alice performs the following steps on the Morello Board:

- 1. **Start:** Alice compiles and runs the programme memory\_reader.py in two different scenarios:
  - Inside the compartment: Alice compiles and runs the integration\_process.c programme on the Morello Board, using the secure environment.
    - Compile: clang-morello -march=morello+c64 -mabi= purecap -g -o integration\\_process integration\ \_process.c -L. -lssl -lcrypto -lpthread
    - Run: proccontrol -m cheric18n -s enable ./
      integration\\_process
  - Outside the compartment: Alice compiles and runs the integration\_process.c programme in the Morello Board's normal operating environment.
    - Compile: clang-morello -o integration\\_process
      integration\\_process.c -lssl -lcrypto -lpthread
    - Run: ./integration\\_process
- 2. **Launch:** Alice starts the script that performs direct memory reading with the following command:
  - Run: python3 memory\\_reader.py
- 3. **Execution:** memory\_reader.py cycles through each RW region, directly reading the data between the start and end addresses of each region. Alice observes the results on the terminal output.

#### Results

The Table 6 shows the results of tests carried out to evaluate access to sensitive data stored and processed by integration\_process running in different environments - inside and outside Morello's secure enclosure. Each test varied the user's permission level (root or with

reduced permissions), recording whether memory access was successful and whether sensitive data was visible.

Table 6	Access contro	I toet	roculte f	or co	ncitivo	data
Table b.	Access contro	ı test	resuus i	or sei	nsilive	gala.

Test	Environment	User Permissions	Access Result	Sensitive Data Visible
1	in Compartment	Root	Success	Yes
2	in Compartment	Different user	Failure	No
3	out Compartment	Root	Success	Yes
4	out Compartment	Different user	Failure	No

Table 6 metadata:

#### **Test ID:**

Unique identification for each test carried out.

#### **Environment:**

Indicates whether the test was carried out inside Morello's secure compartment or in the normal operating environment.

#### **User Permissions:**

Details the user's permission level (e.g., root or a user with reduced permissions).

#### **Access Result:**

Result of the access, indicating whether the memory region was successfully accessed.

#### **Sensitive Data Visible:**

Indicates whether the test was able to visualize sensitive data or not.

# Analysing the results

The graph in the Figure 9 shows the differences in the results of access and visibility of confidential data according to the environment and user permissions.

The results show that access to sensitive data is allowed to any user with root permissions, both in Morello's secure compartment and outside it, indicating that high permissions grant unrestricted access. In contrast, users with reduced permissions are unable to access this data in either environment, demonstrating that the secure enclosure consistently blocks unauthorised access, while the environment outside the enclosure also maintains effective controls.

However, although compartmentalisation is designed to protect sensitive data by restricting access, including to the user who initiated the process, the results indicate that the current configuration may not be sufficient to guarantee complete isolation. This behaviour highlights the importance of investigating further adjustments to the Morello Board's compartmentalisation, especially for contexts where there is a need to protect sensitive data.

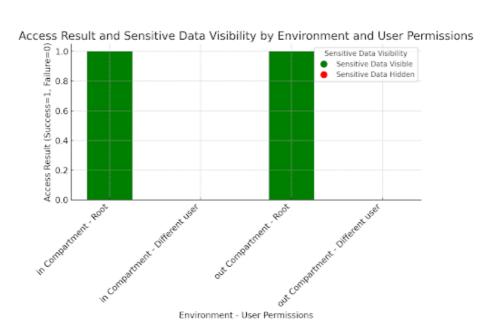


Figure 9. Differences in access results and visibility of confidential data based on environment and user permissions.

# 6. PRINTS

**Process memory regions running outside the secure compartment:** 

P00	TRUTZ	DID PRT	DCC	0000	DEE	90 RAC	- 11	0.71
3781	EX200890	8x282880 FR-	700	~	~5			/home/regis/tee-compartmentalisation-study-case/launcher/programs-data-base/sources/out_server
3701	8x211800	8x214800 r-x8-	- 1	- 1	- 6			/home/regis/bee-compartmentalisation-study-case/launcher/programs-data-base/sources/out_server
3701	8x223880	8x224800 rR-	- 1	- :	- 1			/home/regis/tee-compartmentalisation-study-case/launcher/programs-data-base/sources/out_server
			- 1	- 1	- 1			
3781	8x233880	8x234900 FW-RW	- 1		- 1			
3701	0x48a45660	0x90x29000		٠.		9 6		
3791	0x88a29900	0x80a49000 nx-RW	- 5	. 5	- 4			
3781	0x51071990	6x81872600 f-x	- 3		- 12			
3701	0x85c99900	0x81cc3000 rR-	58	168	18			/usr/11864/118ss1.so.39
3781	6x85cc3866	6x81C62666	- 2			0.01-		
3701	0x81cd2900	0x81d27000 r-xR-		166				/usr/libs4/libsal.so.30
3701	0x81d27900	0x81d36000		•		0.01		
3781	0x80036800	exaudifeee rR-	•					/usr/lib44/libssl.so.30
3701	0x85d3/900	0x81d4e800				0 Oi		
3791	0x8104e000	0x81d54800 nv-RW	- 6					/wsr/11b64/11bss1.so.30
3781	0x12064890	0x52073900 rR-		42	18			/usr/libs4/libthr.so.3
3781	0x82071900	0x12068800				0 Oi		
3701	0x82000000	0x82096800 r-xR-		42	18			\nsr/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
3781	0x52096800	0x52045900				B OI		
3701	0x820a5800	0x120x6000 rR-		•	- 8			/usr/11664/116thr.so.3
3781	0x828a6880	0x82005800				P 01		
3781	0x52065890	0x52067900 nv-RW						/usr/libs4/libthr.so.3
3701	0x82067990	0x820c2000 nv-RW		- 10				
3781	0x82afa880	6x82b77866 r R-	99	392	18			/usr/lib64/libc.se.7
3781	0x82577900	0x82b66600				B OI		
3701	0x82566890	0x82cc2000 r-xR-		392	18			/usr/1/b64/1/bc.se.7
3781	0x82002990	0x82063900				B OI		
3781	0x82cd1900	0x82cdb800 rR-						/usr/lib64/libc.ss.7
3791	0x82c0b800	0x82ces000				# Ot		
3781	0x52cea890	0x52cf3000 nv-RW						/usr/libes/libc.se.7
3781	0x82cf1990	0x82f14800 nx-RW	- 17	17				
3791	0x83076800	0x83703000 r R-	369	848	18			/usr/lib64/libcrysto.se.30
3781	0x53f03800	0x83f13800				B OI		
3791	0x83f13800	0x8411#000 r-xR-			18			/usr/11b64/1Ubcrypto.so.30
3781	6x8411a666	6x84129866				B 01		
3701	0x54129000	0x84176000 rR-						/usr/lib64/libcrypto.so.30
3701	0x8417b800	0x8416a000				# OI		
3781	6x8418a666	6x84192866 nv-8x						/usr/libe4/libcrypto.se.30
3701	0x84192900	0x84194000 nx-RW	- 8	- 3	- 8			
3791	0x84aa6800	0x84sa7800						
3781	0x84f25000	0x84f2a000				B G		
3701	0x8472a890	8x8518a860		•		9 6		
3781	0x85104890	0x85124000 nx-RW						
3701	0x2791127d9000	8x279112767800 rR-	- 3	32	18			/libour/id-alf64.mo.i
3701	0x2791127ee800	6x279112003000 r-xR-		32				/\!hexec/ld-elf64.so.1
3781	0x279112612990	8x279112813880 rR-						/libems/ld-alfe4.so.1
3701	0x279112922990	8x279152923800 nx-RW						/lthmoc/Ld-elf64.mo.1
3701	0x279112823000	8x279112825888 nx-RW						
3781	01305462200000	84305462249800 FW-RW		52				
3701	0x3c5462400000	8x3c5462c4a800 nx-RW	- 34	- 38				
3701	0x1111111111100	0x3000000000000		•		9 6	0	

Figure 10. Process memory regions running outside the secure compartment.

# Memory regions of the process running inside the secure compartment:

regleäta	n:-/performanceTes	ts/security 5 procets END PAT	-v 16	87				
3587	8x100000	END PAT 0x103000 rF	MES	PRES	щ	90	PLAG TP	PATE /home/regis/tee-compartmentalisatios-study-case/launcher/programs-data-base/sources/server
3587	8x103890	0x103000 r1 0x112000			1			
3587 3587	8x112990 8x114990	0x114000 F-x8 0x123000 0x124000 F-x8			-:			/home/regis/tee-compartmentalisatioe-study-case/launcher/programs-data-base/sources/server
3587 3587	8x123880 8x124880	#x124900 r9 #x133900	. 1		-1	- !!	04c vs	/home/regis/tee-compartmentalisation-study-case/launcher/programs-duta-base/sources/server
3587 3587	8x133880 8x134880	8x134900 nx-8 8x135900 nx-8				- 11	c ve	/home/regis/tee-compartmentalisation-study-case/launcher/programs-data-base/sources/server
3587	0x46133860	0x4013d900 r1 0x4014c900	- 10	48	134	62 (	Di 11	/lthexec/ld-elf.sq.1
3587 3587	0x4813d800 0x4814c800	Buddiébada riká	h 31	4	134	12	Di vs	/libenec/ld-elf.so.1
3587	0x4016b000 0x4017a000	0x4017x000 0x4017x000 rx-F			7	1		/libexec/ld-elf.so.i
3587	5x451Te655	PARTICIPAN						
3587 3587	0x4816d800 0x4816e800	0x4015x000 rx-6 0x40192000 rx-6	W 1	1			Dic vs (c se	/libexec/id-elf.so.1
3587	0x48192980 0x48196800	5x40196000 nv-8				- 23	(	
3587	8x48196880	0x4019c000 ru-6 0x401e3000 ru-6	6	192	26	10	01 VI	/vsr/lth/lthost.se.30
3587 3587	0x401x1000 0x401f0000	0x401f0000 0x40244000 r-x5			ı,	10	Di ys	/usr/lth/lthesl.so.30
3587 3587	0x48244890 0x48253800	0x40253800		:	- 1	- 11	04 gd Dic va	/or/(th/thest.ss.30
3587	0x48267990 0x48276890					1	01 01	/mar/tik/tibest.se.30 /mar/tik/tibest.se.30 /mar/tik/tibest.se.30
3587 3587	0x4627f900	Dx49443800 rF	452					
3587 3587	0x48443890 0x48452890	0x49274900 Fu-8 0x49443900 F8 0x49452900 Fx8	467	1121	å	10	04 pd D4 va	/Ith/Uberysto.se.30 /Uh/Uberysto.se.30
3587	0x48525800 0x48634800	0x40534000			•		Di ed	/Uh/Uhoypto.se.30
3587	0x480x5890 0x480x5890	Paradolf Address			į		Di gd	A Company of the Comp
3587 3587	0x48704890	0x49704900 rx-8 6x49767900 rx-8	W 16					/thi/theorypes.se.ue
3587 3587	0x46707990 6x46711990	0x40710000 rw-F 0x40717000 rF	W 7		1	1	Di 10	/usr/(th/cida/(ththr.so.))
3587	0x4671f600 0x4672w600	6x4672x666			•		Di gd	/tit/tiberypto.ss.30 /vsr/tibeci8n/tibetr.ss.3 /vsr/tibeci8n/tibetr.ss.3
3587 3587	0x48744800	0x40744000 r-x5 0x40753000		1			Di 04	/usr/(\b)c18s/(\bthr.so.3 /usr/(\b)c18s/(\bthr.so.3
3587	6x46753860 6x46756860	6x46756600 r8	_		_		DIE VII DI DI	
3587	0x46765890							Approfit Martin (1) before on 3
3587 3587	0x49764890 0x4977b890	0x49756900 rs-9 0x49779900 rs-9 0x49556900 rs-9 0x49516900	17	17 456		2	Di 19	/usr/Ub/c18a/Ubc.so.7
3587 3587	0x4850w980 0x4851d980	0x40016000	305	496			39 VS	/USF/1/10/C189/1/10C.50.7
3587	01/48944660	0x4094e000 r-x3 0x40936000			i		Di gd	/unr/lib/cilin/libc.so.7
3587 3587	0x489756800 0x48975800	0x40975000 r1	- 2				04	/usr/tib/ciss/tibc.so.7 /usr/tib/ciss/tibc.so.7
3587 3587	0x48958880 0x48953880	8x48968888 8x48993888 rw-1 8x48dc4888 rw-1	W 11	22				
3587 3587	8x48dc4889 8x48dc5880	8x48dc5888 8x48de4888 rw-8		1			06 0-c se	
3587 3587 3587	0x48de4890	0x40fe4800 rxx3 0x40ff5800 rxx3	W 34	24			[C 59	
3587	0x48fe4880 0x48ff5880	0x40ffc000	- •	3			04C SW	
3587 3587	0x407fc800 0x41041800	8x41041999 pv-F	# 47	47		*	01c su	
3587	0x41044800	0x41044900	W 25	25	ij		Dic m	
3587 3587	0x41065880 0x4106c880	0x41060900	: :				Se	
3587	0x41075800 0x4107c800	0x4107c000	- •		i	:		
3587 3587	0x4145c990	8x4145c868 8x4147c868 re-1			j		50-c se	
3587 3587	0x4147c990 0x41476990 0x41468990	0x41428000 mx-9 0x41428000 mx-9 0x41464000	W 5	1		- 11	Dic su CC Su	
3587 3587	Grad 5.00048600	8x41496800 nv-6	W 4					
3587 3587 3587	0x414b6890 0x41505880	8x41585888 FW-8	W 65	146 146			Se	
3587	0x41515000	0x41564800 Fa-8 0x41508000 Fa-8	1	146			SW	
3587	0x41500000 0x41500000	0x41a00000 nv-F	W 471	474	- 2		[6 SH [6 SH	
3587 3587	0x41a06660 0x4200660	8x42088880 nx-8	W 319	319	1	- 23	Connect SW	
3587	0x423e0000	9x42499999 nv-6	w 3		1010101		B-c 8	
3587 3587	0x42400000 0x427e0000	8x42Te8880		1	ľ		D-C 19	
3587 3587	0x42c00000 0x42fe0000	0x42fe0000 0x43000000 rx-1	- •				pd 0-c m pd	
3587	0x43000000	0x433e8800	- 1	1	Ť		od	
3587 3587	0x433e0000 0x43400000	0x43400000 nx-9 0x43600000 nx-9 0x43960000 nx-9	w ) w 583	10167	, '	21 .		94
3587 3587	0x43500000 0x43500000	0x43900000 nx-8	W 764	10167	9	21 21	3E	IM Gu
3587	0x43300000 0x44100000	0x43d00000 rx-1 0x44100000 rx-1 0x44200000 rx-1	W 997	10167		21 (	·e	SM
3587 3587	0x44506890 0x44506890	0x4420000 nx-1 0x4420000 nx-1 0x4320000 nx-1 0x4320000 nx-1 0x4710000 nx-1 0x4710000 nx-1 0x420000 nx-1 0x420000 nx-1 0x420000 nx-1 0x420000 nx-1	¥ 1485	10167	í	21 (		94
3587 3587	0x44c00000 0x45300000	0x45300000 rw-8	W 2044	10167		21 21		SM DA
3587 3587 3587	0x4500000 0x46500000	8x46568880 rx-8	W 2559	10167		21 (	c	94
3587 3587	0x47100000	0x47f00000 nx-6	W 3564	10167		28 (		pa .
3587	0x42700000 0x42700000	0x4s100000 rx-1	# 4074 # 5114	10167	3	21 (		
3587 3587	0x4a300000 0x4bbcceso	Buildhood nu-F	W 6128	10167	9	21 (	6	94 94
3587	0x4d700000	0x.45500000 nx-1 0x.46700000 nx-1 0x.46700000 nx-1 0x.54700000 nx-1	W 9124	10157	9	21 (		
3587 3587	6x4F766660 6x52F06660	0x54f00000 nx-8	# 1002 # 1211	5 1806 5 1806	17	21 21	0	SM SM
3587 3587	0x54f00000 0x58f00000	0x50700000 nx-8 0x50700000 nx-9 0x61700000 nx-9	SI 1402 SI 1436	5 1800 9 1804	79 79	21.	0	ga Ga
3587	0x58700000 0x5c700000	0x51700000 re-8	W 1328	10167	9	21 1		MA AND AND AND AND AND AND AND AND AND AN
3587 3587 3587	0xfbfdcescess 0xfbfdcescess 0xffffbfeffsss	exforcessesso ru- exforcessosso ru- exfff/bffsesso ru-	848	1			Su Di Su	
3587	Oxffffbfwffboo Oxffffbffbeboo	Bufffffffgggg ru-F			1		Dic m	
3587 3587 3587	Oufffffffceeso Ouffffffffceeso	8xfffffff68800 ru-1 0x280088008800 ru-1			1		04 	
2001	Section (TTT Code)	Management [-X-	- 1		- 44		pa	

Figure 11. Memory regions of the process running inside the secure compartment.

# Finalisation of the process by the system

The process runs for a certain period of time (around 1 hour) and is finalised by the system: killed.

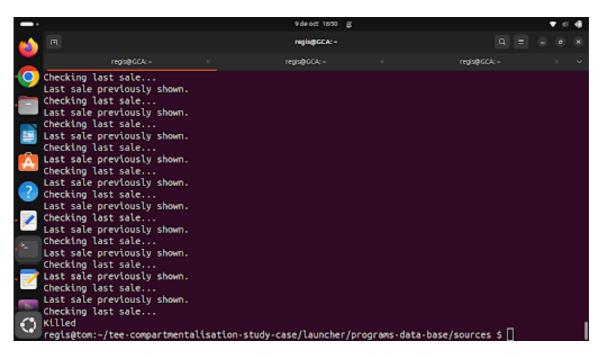


Figure 12. Finalisation of the process by the system.

# The morello board crashed at this point.

Regis needs to contact the morello-board s administrator [erik] to reboot.

3587	0x43000000	0x433e0000		Θ	Θ	0	0 G gd
3587	0x433e0000	0x43400000	rw-RW	3	3	1	0D-c sw
3587	0x43400000	0x43680000	rw-RW	583	101679	21	θc sw
3587	0x43680000	0x43980000	rw-RW	764	101679	21	θc sw
3587	0x43980000	0x43d00000	rw-RW	882	101679	21	0c sw
3587	0x43d00000	0x44100000	rw-RW	997	101679	21	0c sw
3587	0x44100000	0x44600000	rw-RW	1277	101679	21	Θc sw
3587	0x44600000	0x44c00000	rw-RW	1485	101679	21	0c sw
3587	0x44c00000	0x45300000	rw-RW	1788	101679	21	0c sw
3587	0x45300000	0x45b00000	rw-RW	2044	101679	21	0c sw
3587	0x45b00000	0x46500000	rw-RW	2559	101679	21	0c sw
3587	0x46500000	0x47100000	rw-RW	3065	101679	21	ΘC SW
3587	0×47100000	0x47f00000	rw-RW	3584	101679	21	ΘC SW
3587	0x47f00000	0x48f00000	rw-RW	4074	101679	21	θc sw
3587	0x48f00000	0x4a300000	rw-RW	5114	101679	21	θc sw
3587	0x4a300000	0x4bb00000	rw-RW	6120	101679	21	0c sw
3587	0x4bb00000	0x4d700000	rw-RW	7163	101679	21	θc sw
3587	0x4d700000	0x4f700000	rw-RW	8124	101679	21	0c sw
3587	0x4f700000	0x51f00000	rw-RW	10225	10167	9 21	θc sw
3587	0x51f00000	0x54f00000	rw-RW	12119	9 101679	9 21	θc sw
3587	0x54f00000	0x58700000	rw-RW	14025	101679	9 21	0c sw
3587	0x58700000	0x5c700000	rw-RW	14359	10167	9 21	θc sw
3587	0x5c700000	0x61700000	rw-RW	1328	101679	21	0c sw
3587	0xfbfdbffff000	0xfbfdc0000000	ſ₩	1	Θ	1	0 Cc sw
3587	0xfbfdc0000000	0xfe0000000000		848	Θ	-	0 C sw
3587	0xffffbfeff000	0xffffbff80000	rw-RW	1	1		0 CNc sw
3587	0xffffbff80000	0xffffffff60000		Θ	Θ	0	0 G gd
3587	0xfffffff60000	0xffffffff80000	rw-RW	3	3	1	0 CD-c sw
3587	0xfffffffff000	0×100000000000000	r-x	1	1 4	44	0 ph

Figure 13. The morello board crashed at this point.

The remote ssh shell that connects to the morello board crashes when the mem scanner prog tries to read this range of mem add: 0x4a300000 - 0x4bb00000.

Other ssh shell connections continue working as normal.

There are occasion when the prog manage to crash the whole morello board [the actual cheriBSD], no more shells can be opened. It seem that total crash takes place when the prog tries to read some specific mem ranges [we dont know which ones cause total crashes; we believe that they are ranges where priviledge soft runs.

This is our preliminary observation.

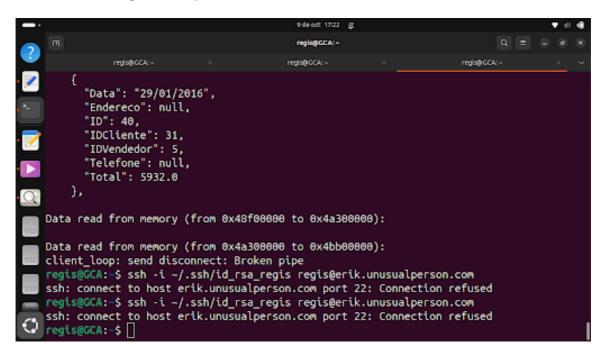


Figure 14. client\_loop: send disconnect: Broken pipe.

# Error occurs after server crash and restart [Erik].

#### **Error accessing process memory:**

```
[Errno 2] No such file or directory: '/proc/3587/mem'
```

This error occurs because the file /proc/{pid}/mem, which the script tries to access to read a process's memory, is not available or cannot be accessed. This may happen if the process does not exist, the PID is incorrect, or the script does not have the necessary permissions to access this path.

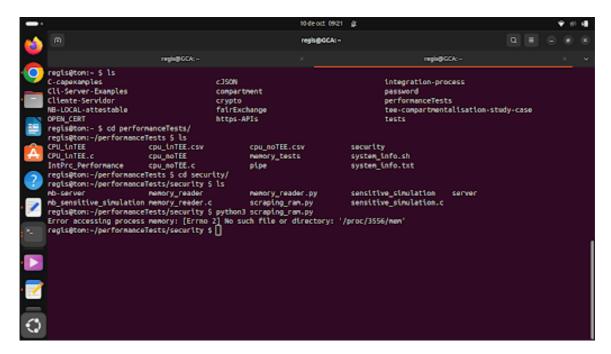


Figure 15. [Errno 2] No such file or directory: '/proc/3587/mem'.

#### Procedure for running memory\_reader.py after MB is rebooted.

Check that /proc is mounted correctly:

```
regis@tom:~/performanceTests/security $ mount | grep /proc
```

Figure 16. Check that /proc is mounted correctly.

If it's not mounted, you need to try mounting it:

```
regis@tom:~/performanceTests/security $ sudo mount -t procfs proc /proc
```

Figure 17. If it's not mounted, you need to try mounting it.

After the command to mount /proc, it was simply possible to run the memory\_reader.py script again.