ELE3001 BEng (Hons) Final Year Project Report

Applicability of Digital Security by Design to Performance-Sensitive Networking Applications

Michael Allen
40266651

17 April 2023

Abstract

Packet processing applications typically interact with untrusted, third-party plugins, introducing a trade-off between security and performance. It was previously theorised that this trade-off can be optimised if Digital Security by Design (DSbD) principles are implemented for communications between the application and plugins.

This report shows the creation of a packet processing application on the Morello board, which implements both DSbD principles and a traditional inter-process communications (IPC) methodology. This application is built using the Data Plane Development Kit (DPDK) framework. It is demonstrated that DSbD initiatives can provide the required security for communications between the application and plugins. It is also shown that the use of DSbD concepts in the packet processing application improves the performance over IPC, regarding packet processing latency, total CPU utilisation, and total execution time. This behaviour is confirmed for a variety of packet streams, with packet counts ranging from 20,000 to 200,000 and packet sizes from 512B to 8KiB.

Specification

Electrical & Electronic Engineering, Software & Electronic Systems Engineering
Final Year Projects 2022-2023

Applicability of DSbD to Performance-Sensitive Networking Applications

Supervisor: Sandra Scott-Hayward

	Control	x	Embedded Systems		High Frequency Electronics		Microelectronics
	Electric Power	x	Software		Connected Health		MEMS
x	Cyber-Security	x	Wireless Communications		Signal/Image Processing		Intelligent Systems
	Digital Design		Sensor Networks	x	Data Analytics		Electronics

The area of investigation is the application of DSbD technologies to the performance-sensitive distribution of data from a broker to untrusted consumer plugins. A concrete example of this is in a networking application or appliance (such as a firewall or network monitor) which receives packets from an incoming network device, classifies the packets and distributes them to one or more '3rd-party' untrusted plugins for consumption.

Such a program structure entails an undesirable trade-off between performance (co-located plugins within the main address space rather than separate plugin processes with consequent IPC overhead) and security (isolated address spaces for untrusted code vs their inclusion in the address space of the main process, which may allow them unauthorised access to additional data).

We would like to demonstrate that DSbD can permit the elimination of this trade-off and enable an architecture which is both secure and performant.

Objectives

1. Familiarisation with Morello, CheriBSD, and CHERI concepts.
2. Successfully boot, install, and configure CheriBSD on the supplied Morello board.
3. Port a packet processing library to run with CHERI capabilities enabled on the Morello board.
4. Development of a packet processing application for CheriBSD and the Morello board, which can use both CHERI-enabled in-process plugins ("DSbD design") and a traditional multi-process structure.
5. Add protection to DSbD design (bounds checking and appropriate permissions) and demonstrate that it works (secure).
6. Define simple packet streams (including a range of packet sizes) and transmit those streams to both versions of the packet processing application.
7. Measure and analyse key performance characteristics including packet processing latency and CPU utilisation for both versions and write these up in a final report.

Learning Outcomes

Upon completion of the project you will expect to have:

1. Hands-on knowledge of CHERI including CheriBSD.
2. Enhanced C and systems/embedded programming capability.
3. Ability to identify, measure, record and interpret key performance metrics.

Acknowledgements

I would like to thank my project supervisor, Sandra Scott-Hayward, for providing high-quality, frequent feedback on my work and offering encouragement throughout. To Gareth Conway, for supervising the end of this project and supporting me. To my boss at Pytilia, Tim Silversides, for conceptualising this project and helping me learn the required fundamental concepts. To my colleague, Neil Sinclair, for accompanying me to the Digital Catapult demonstration day and delivering a presentation on my work. To my colleague, Eileen O'Reilley, for assisting me with porting DPDK to the Morello board. To the Digital Catapult team, for providing the Morello board used in this project and offering a wide range of useful learning resources related to DSbD.

Declaration of Originality

By submitting this report electronically or physically to the School of EEECS for assessment I declare that:

- I have read the University regulations on plagiarism, and that the attached submission is my own original work except where clearly identified.
- I have acknowledged all written and electronic sources used.
- I agree that the School may scan the work with plagiarism detection software.

Signed: Michael Allen

Date: 17/04/2023

Contents

List of Abbreviations

CHERI	Capability Hardware-Enhanced RISC Instructions
CLI	Command Line Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
DHCP	Dynamic Host Configuration Protocol
DPDK	Data Plane Development Kit
DSbD	Digital Security by Design
FVP	Fixed Virtual Platform
GDB	GNU Debugger
IP	Internet Protocol
IPC	Inter-Process Communications
ISA	Instruction Set Architecture
LAN	Local Area Network
OS	Operating System
pcap	Packet Capture
PureCap	Pure Capability
QUB	Queen’s University Belfast
RISC	Reduced Instruction Set Computing
SoC	System on Chip
SSH	Secure Shell
TAP	Technology Access Programme
TCP	Transmission Control Protocol
TLS	Thread Local Storage
UDP	User Datagram Protocol
USB	Universal Serial Bus

1 Introduction

1.1 Background

Digital Security by Design (DSbD) is a growing cyber-security initiative backed by the UK government. The aim of DSbD is to fundamentally redesign the digital computing infrastructure in a more secure manner. This involves building memory access limitations into the infrastructure in a way which reduces the risk of vulnerabilities and minimises their effect [1]. The main technology used to accomplish this goal is called capability hardware-enhanced RISC instructions (CHERI).

Packet processing is the application of various algorithms to a packet of data during its transmission through a network [2]. A wide variety of applications use packet processing, including firewalls [3], network monitors [4], and storage backup [5]. These applications commonly use third-party plugins to provide additional functionality. For example, firewall plugins may provide user authentication, spam or bot detection, and region blockers [6] [7].

A third-party plugin can come from any source, so it can contain malicious or poorly written code. Therefore, a trade-off between performance and security must be made to allow an application to interact with such a plugin [8]. If a plugin has access to the same virtual address space as the application, then only a pointer to packet data is required by the plugin. This is highly performant but introduces a security risk because the untrusted plugin will receive full access to the application memory [8]. Alternatively, if a plugin has a separate virtual address space to the application, then it requires a copy of the packet data to be sent through inter-process communications (IPC) [9]. This provides security because the plugin does not have access to the application memory. However, the use of IPC introduces a performance penalty [8].

1.2 Approach

This report investigates the applicability of DSbD initiatives to performance-sensitive networking applications.

This investigation was completed by first setting up a Morello board and installing the CheriBSD operating system (OS). The setup was validated by creating sample C applications. The Data

Plane Development Kit (DPDK) framework was then ported to the Morello board with CHERI capabilities enabled.

Using this DPDK port, a packet processing application was developed to interact with two untrusted, third-party consumer plugins. This application can dispatch packets to these consumers through either CHERI capabilities or a traditional IPC methodology or can read packets and not dispatch them to any consumers. The CHERI capabilities are restricted programmatically and these restrictions were shown to be sufficiently secure through appropriate tests. The IPC dispatch mode communicates with an external packet receiver application which was also developed for this project.

A Python script was created to generate a series of packet streams which varied in both packet size and count. These streams were used as input to the packet processing application to allow performance metrics to be recorded across a range of different inputs for each method of packet dispatch. The performance metrics recorded were packet processing latency, total central processing unit (CPU) utilisation, and total application execution time. These metrics were recorded 10 times for each packet stream, allowing a mean and standard deviation for each stream to be calculated and plotted on appropriate graphs. The metrics recorded for the processing mode which does not dispatch packets provide a set of control measurements to compare the other dispatch methods against. The graphs were analysed to determine what benefits or drawbacks are provided to networking applications by DSbD initiatives.

Finally, there is a discussion on the overall work completed for this project. This includes the conclusions reached, an analysis of work completed against the project objectives and planned timeline, the response from the industrial partners, and the potential future work relating to this project that could be undertaken.

1.3 Industrial Partners

This report relates to a final year project completed for Queen's University Belfast (QUB). However, there are 2 industrial partners associated with this project. The digital and software industry innovation agency for the British government, Digital Catapult [10], conducted this project as part of cohort 2 of the DSbD Technology Access Programme (TAP). The project outline was created by Pytilia, a Belfast-based software solutions company [11]. Pytilia conducted this project in association with QUB and Digital Catapult.

1.4 Previous Work

Pytilia worked with DSbD in 2021 as part of the Software Ecosystem competition [8]. This resulted in a port of the DPDK framework to the Arm Morello Fixed Virtual Platform (FVP) [12].

A packet processing application, Limelight, was developed for the Morello FVP using the aforementioned DPDK port. This application could read packets from a packet capture (pcap) file and print the packet data to the terminal. This was the first project to show that CHERI capabilities could be used in conjunction with packet processing [8].

The Limelight project was successful, but left room for future development. Firstly, the application did not contain functionality to dispatch any packets so it did not interact with any packet consumers. Secondly, there were no performance metrics recorded to evaluate what effect DSbD principles had on performance. Thirdly, the Limelight application could only read up to 16 packets in total before raising an error.

2 Theory

This section covers the theory relating to this project. This includes details on various DSbD principles and the DPDK framework. This relates to task 1 of the project timeline shown in Appendix A and fulfils project objective 1 from the specification. This section expands upon the background given in the interim report for this project [13].

2.1 Digital Security by Design

The underlying technology behind DSbD is called capability hardware-enhanced RISC (reduced instruction set computer) instructions, or CHERI. This technology has been developed by the University of Cambridge and SRI International. CHERI replaces traditional pointers to virtual memory addresses with architectural capabilities, containing a virtual address and corresponding metadata [14]. This is depicted in Figure 2.1 below.

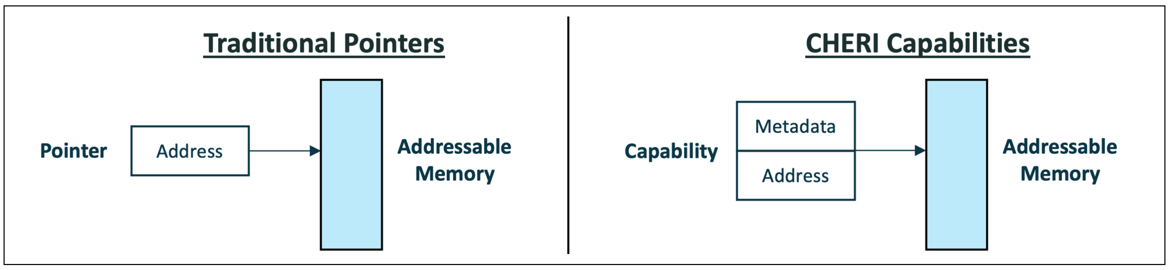


Figure 2.1: Comparison of pointers and CHERI capabilities

The metadata associated with a CHERI capability provides built-in protection to the memory address in the following ways [15]:

- **Bounds** - Specifies the range of addresses that the capability has access to.
- **Permissions** - Specifies what access rights are granted to the capability (read, write, execute).
- **Monotonicity** - Ensures the capability bounds cannot be increased and permissions cannot be added.
- **Provenance Validity** - Ensures the capability was derived from an existing capability through a valid method.

CHERI capabilities can replace pointers in the C and C++ programming languages which do not have intrinsic memory security. This means that certain software runtime security checks can be removed, improving performance [16].

Another security feature which CHERI implements is called software compartmentalisation. This involves splitting one monolithic process into multiple smaller compartments. This reduces the impact of a software vulnerability because only the affected compartment can be exploited, while all other compartments within the process remain secure [17].

Morello is a research programme conducted by Arm to redesign computer processors in a way which is less vulnerable to security breaches. This is accomplished through extending RISC instruction set architectures (ISAs) to implement CHERI concepts [18]. The Morello fixed virtual platform (FVP), released in October 2020, is an emulator which implements the Morello architectural concepts [19]. The Morello board, also known as the Morello development board or Morello evaluation board, is a prototype of the physical Morello architecture containing a system on chip (SoC) that implements CHERI concepts, which was released in January 2022 [14].

CheriBSD is an OS adapted from FreeBSD to support the Morello architecture. It is Unix-based and enables CHERI capabilities in both the kernel and userspace. This allows for capability bounds checking, permission enforcement, and software compartmentalisation [20].

The CHERI LLVM project is a toolchain which facilitates the compilation of CHERI-enabled C and C++ code. It extends the LLVM toolchain to facilitate CHERI capabilities. It includes the CHERI Clang compiler, optimisers, linkers, and C/C++ libraries [21].

There are 2 compilation modes which can be used by the CHERI Clang compiler [22]:

- **Hybrid** - Traditional machine-word pointers are used by default when declaring a pointer. However, a tag can be specified to use a CHERI capability instead.
- **Pure Capability (PureCap)** - Every pointer is replaced by a CHERI capability and it is impossible to declare a traditional pointer.

The CHERI GNU Debugger (GDB) is a debugger which is compatible with CHERI-enabled C/C++ code. It extends GDB to allow for debugging of compiled CHERI C/C++ applications. This means that capability bounds and permissions are shown and capability validation errors are correctly displayed to the user [23].

2.2 Data Plane Development Kit

DPDK is a packet processing framework which was released in 2010 by Intel. It is open-source and written in C [24]. It is widely used in industry because of its high performance, and receives regular updates from contributors including Intel, IBM, and Cisco [25]. DPDK is compatible with a wide array of hardware architectures produced by different companies, including Arm [26]. DPDK supports Windows [27], FreeBSD [28], and a variety of Linux distributions [29]. In 2021, Pytilia ported DPDK (v20.11.1) to the Morello FVP, which used a custom build of the CheriBSD OS [12].

2.3 Progress Reflection

Task 1 of the project timeline shown in Appendix A refers to research into the theory and background related to this project. 4 weeks were allocated to this task, split between DSbD concepts and packet processing. The research required to begin work on this project was completed in the allocated time. However, more research was conducted throughout the project which helped the development progress.

Objective 1 of the project specification requires familiarisation with DSbD concepts, specifically CHERI, CheriBSD, and Morello. This objective is fulfilled in this section.

3 Setup

This section covers the setup procedure of the Morello board, including the steps taken for setting up both the hardware and software. This relates to task 2 of the project timeline shown in Appendix A and fulfils project objective 2 from the specification. This section expands upon the setup process detailed in the interim report for this project [13].

3.1 Hardware

The Morello board was delivered before the project start date. The board was prebuilt so the only hardware setup involved connecting the power and Ethernet cables. No peripheral devices were supplied with the Morello board. The connected Morello board is shown in Figures 3.1 and 3.2 below.

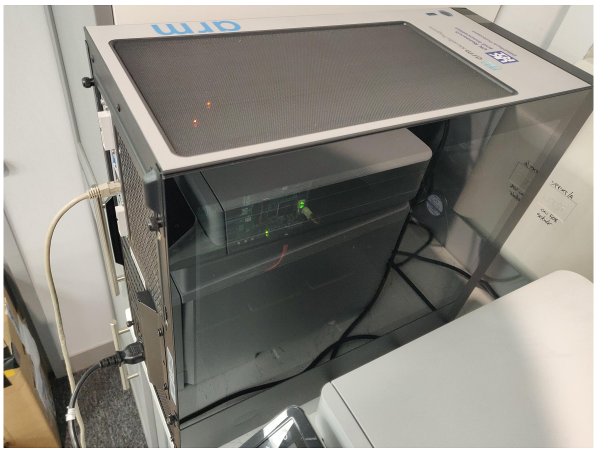


Figure 3.1: Side view of the Morello board.

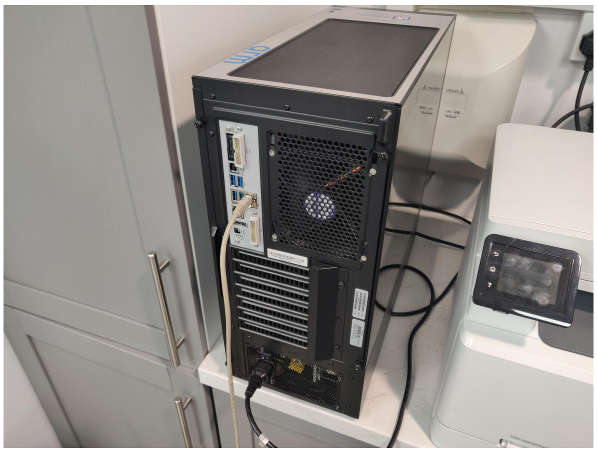


Figure 3.2: Back of the Morello board.

3.2 Software

The Morello board did not have any preinstalled software. Therefore, the first step in setting up the board was installing an OS. The CheriBSD was required by the project specification because it facilitates the use of CHERI capabilities. A prebuilt disk image of `CheriBSD 22.05p1` was sourced from the CheriBSD website [20], which was the most recent official release at the time of installation. A portable universal serial bus (USB) device was flashed with this image to create a bootable USB drive.

Due to the lack of peripheral devices, the Morello board was connected to a MacBook Pro through a debug USB connection when installing CheriBSD. The MacBook used the `screen` terminal emulator for communications with the Morello board. This emulator was recommended by the Morello getting started guide [30].

Various tasks were completed after CheriBSD was installed, including:

- Configuration of the date and time.
- Creation of a local user to avoid using the root user.
- Connection of an Ethernet cable to establish network connectivity.
- Reservation of an internet protocol (IP) address in the local area network (LAN) via dynamic host configuration protocol (DHCP) reservation.
- Installation of CHERI-enabled packages, including `Vim`, `Python`, and `cscope`.
- Replacement of the default terminal shell, `sh`, with `Bash` to provide more functionality.
- Creation of a `vimrc` configuration file to improve the usability of the `Vim` text editor.

Once network connectivity was established on the Morello board, a secure shell (SSH) key was generated. This allowed a remote connection to be established between the Morello board and an external computer in the same LAN. However, remote access to the Morello board could not be achieved outside of the LAN it was connected to. This also provided automatic authentication when connecting to remote git repositories hosted on GitLab and GitHub.

3.3 Progress Reflection

Task 2 of the project timeline shown in Appendix A covers the setup process, which was allocated 2 weeks. The first week was scheduled for booting the Morello board. However, the Morello board unexpectedly arrived prebuilt, so this was completed in less than a day. The installation of CheriBSD on the Morello board took approximately 2 weeks. This was due to an unsuccessful attempt to build a custom image of CheriBSD before the prebuilt image of `CheriBSD 22.05p1` was discovered. Furthermore, network connectivity took another week to establish because of an unsuccessful attempt to configure a Wi-Fi connection before deciding to use Ethernet. The other tasks listed in this setup process were relatively quick to complete and did not significantly impact the duration of this task. Overall, this task exceeded the expected duration by a week.

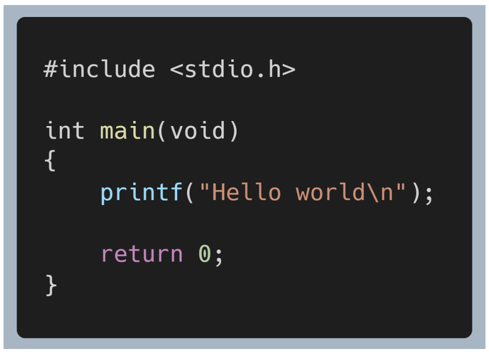
Objective 2 of the project specification asks for CheriBSD to be installed and configured on the Morello board. This section fulfils that objective.

4 C Applications

This section covers the creation of basic C applications on the Morello board. This relates to task 3 of the project timeline shown in Appendix A. This section expands upon the C applications detailed in the interim report for this project [13].

4.1 Helloworld

The first C application created on the Morello board was a Helloworld program. The application was designed to print the phrase "Hello world" to the terminal by using the `printf` function. The code for this application was written in a file called `helloworld.c` and is shown in Figure 4.1 below.



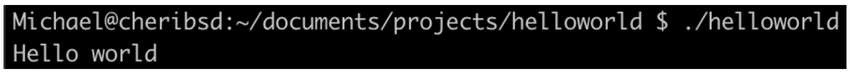
```c
#include <stdio.h>

int main(void)
{
    printf("Hello world\n");

    return 0;
}
```

Figure 4.1: The C code used for the Helloworld application.

The application was compiled in both hybrid and PureCap modes to test the functionality of the CHERI Clang compiler. The following terminal commands were used to compile in these modes:

- **Hybrid:** `cc -o helloworld helloworld.c -mabi=aapcs`
- **PureCap:** `cc -o helloworld helloworld.c -mabi=purecap`

The corresponding command used to run the compiled application is `./helloworld` for both versions. Figure 4.2 below shows the output produced by this application when executed from the terminal.



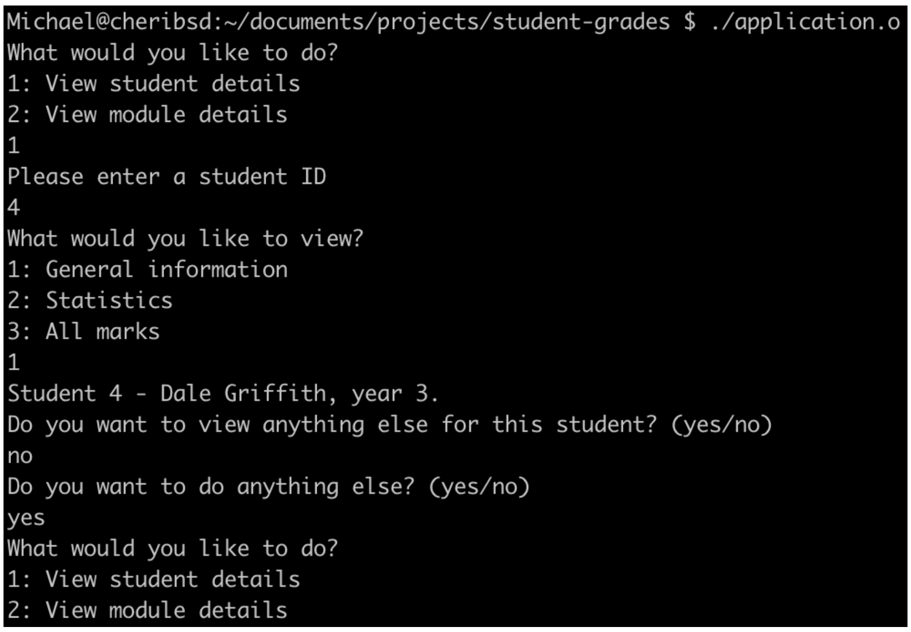
```
Michael@cheribsd:~/documents/projects/helloworld $ ./helloworld
Hello world
```

Figure 4.2: Output of the C Helloworld application.

As shown in Figure 4.2, the application outputs the desired phrase of “Hello world”. This output was identical for both versions of the application. This proved that the CHERI Clang compiler was correctly installed and configured to allow both hybrid and PureCap modes of compilation.

4.2 Student Information

A more advanced C application was created on the Morello board which managed student information. This application was compiled in PureCap using the CHERI Clang compiler. The application received details on students, grades, and modules from respective comma-separated value (CSV) files. The code was organised into separate C header/source files. User input from the terminal was used to determine what data to display, including student information, module descriptions, and grade statistics. An example usage of this application can be seen in Figure 4.3 below.



```
Michael@cheribsd:~/documents/projects/student-grades $ ./application.o
What would you like to do?
1: View student details
2: View module details
1
Please enter a student ID
4
What would you like to view?
1: General information
2: Statistics
3: All marks
1
Student 4 - Dale Griffith, year 3.
Do you want to view anything else for this student? (yes/no)
no
Do you want to do anything else? (yes/no)
yes
What would you like to do?
1: View student details
2: View module details
```

Figure 4.3: The student information application.

Figure 4.3 shows that the application output the results which correspond to the terminal input given by the user. The successful execution of this application proved the following points:

- The linker was configured correctly because the external C header and source files were imported without error.
- Terminal input is read correctly on the Morello board.

- File input and output on the Morello board was possible because the application could read from a CSV file.
- A `Makefile` can be used with the CHERI Clang compiler and the `make` command can be used to complete a build.

The development of this application revealed that the built-in C function `memcpy` was unreliable when using CHERI capabilities. This is because it was prone to capability bounds errors due to buffer overruns. The `strcpy` function was determined to be a reliable replacement because it did not raise these errors.

The code for this application was added to a git repository and pushed upstream to GitLab [31] and GitHub [32].

4.3 Progress Reflection

Task 3 of the project timeline shown in Appendix A relates to the creation of basic C applications. This was allocated 3 weeks in total. However, this task was completed quicker than expected, requiring only 2 weeks. This compensated for the extra week required in the previous task, setting the project back on schedule.

Building these applications proved the Morello board was configured to provide the required functionality to continue this project.

5 DPDK Port

This section covers the porting process of the DPDK framework, including the required code changes to compile the framework and the limitation of the port to a single process. This relates to task 4 of the project timeline shown in Appendix A and fulfils project objective 3 from the specification. This section expands upon the porting process detailed in the interim report for this project [13].

5.1 Overview

The packet processing framework, DPDK, was chosen to be ported to the Morello board after a discussion with Pytilia. This was agreed upon because Pytilia had previously completed a port of DPDK to the Morello FVP. This port was forked from DPDK version 20.11.7 and pushed upstream to GitHub [12]. This was used as the base for the port to the Morello board because of the architectural similarities between the Morello board and FVP.

The DPDK build process requires 2 build steps, which are:

- **The Meson build** - This build step is completed inside the DPDK root directory with the `meson setup {build_dir}` command, where `build_dir` is the target build directory. This handles the hardware-specific configuration for the build process and produces the required build directory.
- **The Ninja build** - This build step is completed inside the build directory generated by the Meson build with the `ninja` command. This handles the compilation and linking of the C code, including libraries, network drivers, and example networking applications.

5.2 Code Changes

This section details the required code changes which were made to the DPDK source code to allow the Meson and Ninja builds to complete without error.

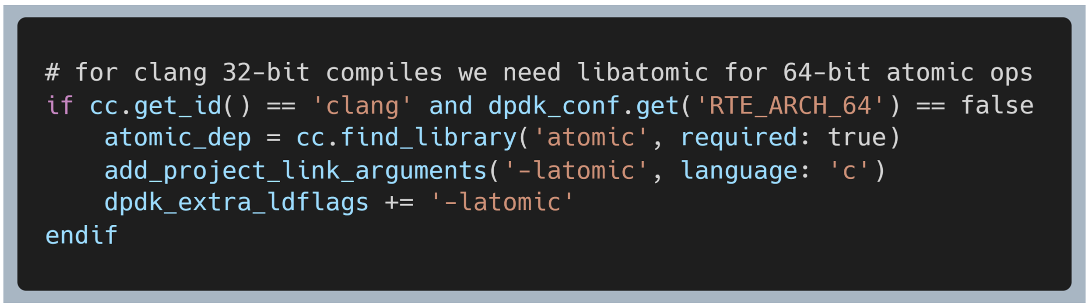
5.2.1 Compiler Flags

To compile the DPDK library for the Morello board, the compiler flags had to be updated. The following changes were made to the `meson.build` configuration file located in the `config/arm` subdirectory of the DPDK framework:

- The compiler flag `-march=armv8-a+crc` was replaced with `-march=morello+c64` to enable compilation with the Morello architecture.
- The compiler flag `-mabi=purecap` was added to compile DPDK in PureCap mode. This ensures that CHERI capabilities are always enabled as per objective 3 of the project specification.

5.2.2 Atomic Dependency

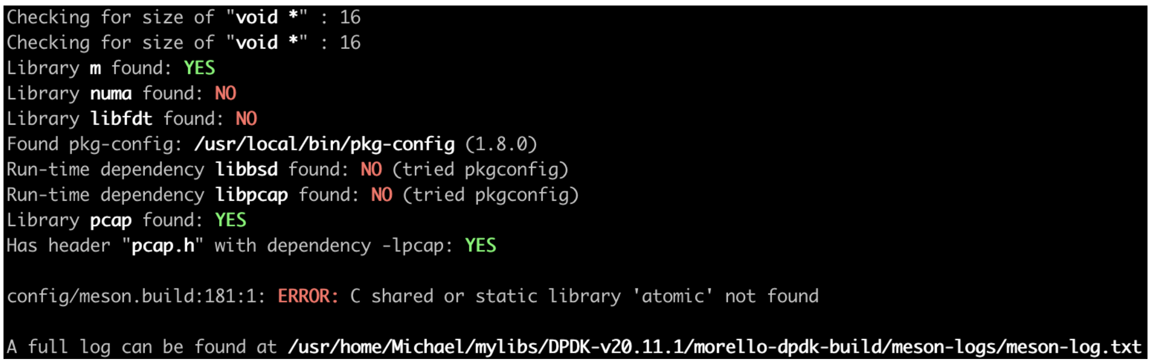
The Meson build would not complete because the `atomic` library was not installed on the Morello board. This library is incompatible with the Morello board so the issue could not be resolved by installing it. However, the DPDK source code shows that the `atomic` library is only required for 32-bit machine architectures. The code used to import the `atomic` library is shown in Figure 5.1 below.



```
# for clang 32-bit compiles we need libatomic for 64-bit atomic ops
if cc.get_id() == 'clang' and dpdk_conf.get('RTE_ARCH_64') == false
    atomic_dep = cc.find_library('atomic', required: true)
    add_project_link_arguments('-latomic', language: 'c')
    dpdk_extra_ldflags += '-latomic'
endif
```

Figure 5.1: Code snippet which imports the `atomic` library.

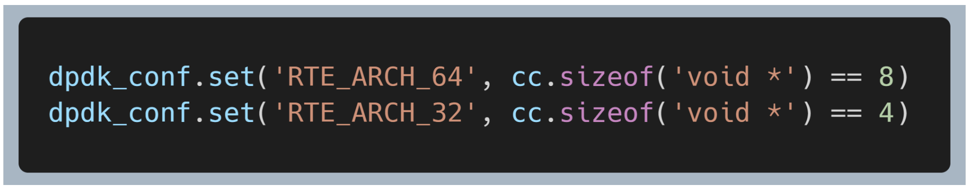
The Morello board uses a 64-bit CPU, so there should not be an attempt to import the `atomic` library. However, a corresponding error was raised during the Meson build, as shown in Figure 5.2 below.



```
Checking for size of "void *" : 16
Checking for size of "void *" : 16
Library m found: YES
Library numa found: NO
Library libfdt found: NO
Found pkg-config: /usr/local/bin/pkg-config (1.8.0)
Run-time dependency libbsd found: NO (tried pkgconfig)
Run-time dependency libpcap found: NO (tried pkgconfig)
Library pcap found: YES
Has header "pcap.h" with dependency -lpcap: YES

config/meson.build:181:1: ERROR: C shared or static library 'atomic' not found

A full log can be found at /usr/home/Michael/mylibs/DPDK-v20.11.1/morello-dpdk-build/meson-logs/meson-log.txt
```

Figure 5.2: The `atomic` dependency error raised during the Meson build.

As shown in Figure 5.2, the Meson build checked the size of the `void *` data type twice. The size is shown to be 16, which is measured in bytes. The Meson build uses this size to determine whether the CPU uses 32 or 64-bit virtual memory addresses. Therefore, the code is executed once when checking if the CPU is 64-bit and again when checking if the CPU is 32-bit. The corresponding code for this check is shown in Figure 5.3 below.



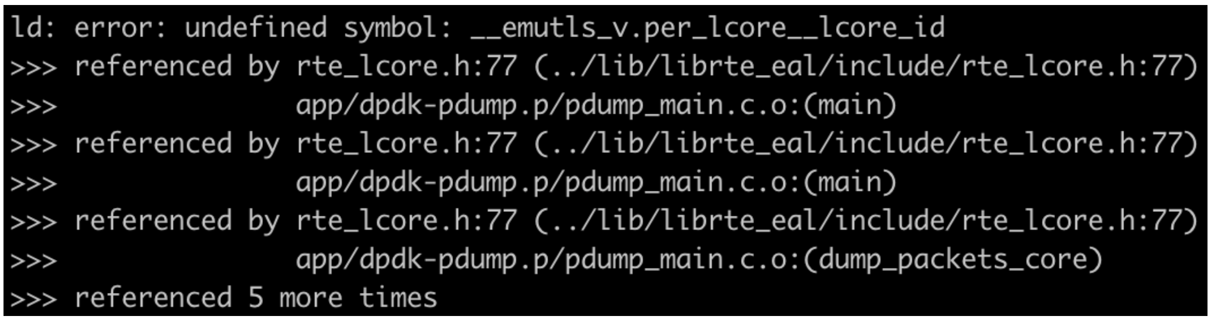
```
dpdk_conf.set('RTE_ARCH_64', cc.sizeof('void *') == 8)
dpdk_conf.set('RTE_ARCH_32', cc.sizeof('void *') == 4)
```

Figure 5.3: Code snippet which determines the CPU architecture type.

Figure 5.3 shows that a CPU is determined to be 64-bit if the size of the `void *` data type is 8 bytes and 32-bit if the data type requires 4 bytes. This check works on most architectures because `void *` refers to a pointer to a virtual address. However, on the Morello board, this data type refers to a CHERI capability, which contains both an address and metadata. A CHERI capability requires double the storage space of a pointer, so for a 64-bit architecture it requires 16 bytes and for a 32-bit architecture it requires 8 bytes. The Morello board has a 64-bit CPU so the size is measured as 16 bytes. Therefore, both checks fail to detect the correct architecture and the default setting of 32-bit is used. To fix this issue, the checks were updated according to the size of CHERI capabilities. Fixing this error allowed the Meson build to complete.

5.2.3 Thread Local Storage

The Ninja build raised an error because emulated thread local storage (TLS) could not be used on the Morello board. This error is shown in Figure 5.4 below.



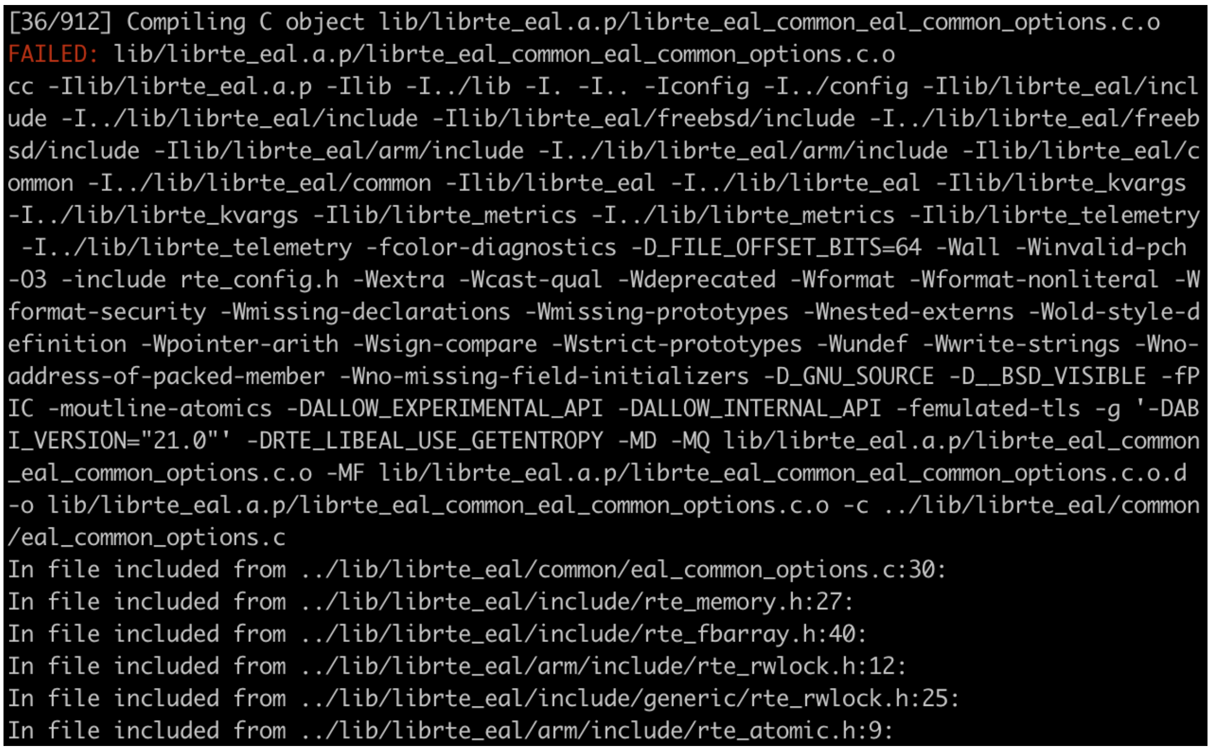
```
ld: error: undefined symbol: __emutls_v.per_lcore__lcore_id
>>> referenced by rte_lcore.h:77 (../lib/librte_eal/include/rte_lcore.h:77)
>>>               app/dpdk-pdump.p/pdump_main.c.o:(main)
>>> referenced by rte_lcore.h:77 (../lib/librte_eal/include/rte_lcore.h:77)
>>>               app/dpdk-pdump.p/pdump_main.c.o:(main)
>>> referenced by rte_lcore.h:77 (../lib/librte_eal/include/rte_lcore.h:77)
>>>               app/dpdk-pdump.p/pdump_main.c.o:(dump_packets_core)
>>> referenced 5 more times
```

Figure 5.4: The emulated TLS error raised during the Ninja build.

To fix the error shown in Figure 5.4, the compiler flag `-femulated-tls` was removed. This meant that the code was compiled to use physical TLS instead. This compiler flag was added for the DPDK port to the Morello FVP because it is an emulator, so it could not use physical TLS. However, this change is incompatible with a physical CPU such as the Morello board, so it was reversed.

5.2.4 Incompatible Drivers

The Ninja build raised an error related to the use of atomic operations in some of the drivers. This error is shown in Figure 5.5 below.



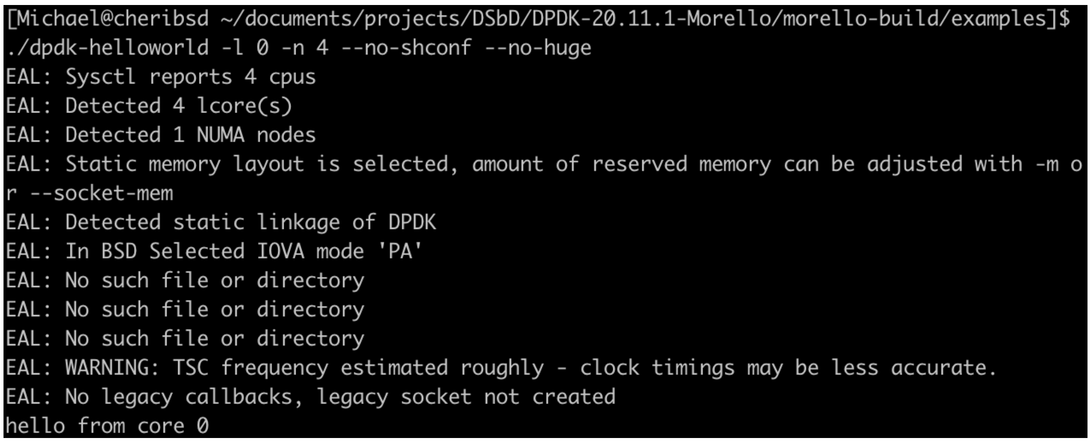
```
[36/912] Compiling C object lib/librte_eal.a.p/librte_eal_common_eal_common_options.c.o
FAILED: lib/librte_eal.a.p/librte_eal_common_eal_common_options.c.o
cc -Ilib/librte_eal.a.p -Ilib -I../lib -I. -I.. -Iconfig -I../config -Ilib/librte_eal/incl
ude -I../lib/librte_eal/include -Ilib/librte_eal/freebsd/include -I../lib/librte_eal/freeb
sd/include -Ilib/librte_eal/arm/include -I../lib/librte_eal/arm/include -Ilib/librte_eal/c
ommon -I../lib/librte_eal/common -Ilib/librte_eal -I../lib/librte_eal -Ilib/librte_kvargs
-I../lib/librte_kvargs -Ilib/librte_metrics -I../lib/librte_metrics -Ilib/librte_telemetry
 -I../lib/librte_telemetry -fcolor-diagnostics -D_FILE_OFFSET_BITS=64 -Wall -Winvalid-pch
-O3 -include rte_config.h -Wextra -Wcast-qual -Wdeprecated -Wformat -Wformat-nonliteral -W
format-security -Wmissing-declarations -Wmissing-prototypes -Wnested-externs -Wold-style-d
efinition -Wpointer-arith -Wsign-compare -Wstrict-prototypes -Wundef -Wwrite-strings -Wno-
address-of-packed-member -Wno-missing-field-initializers -D_GNU_SOURCE -D__BSD_VISIBLE -fP
IC -moutline-atomics -DALLOW_EXPERIMENTAL_API -DALLOW_INTERNAL_API -femulated-tls -g '-DAB
I_VERSION="21.0"' -DRTE_LIBEAL_USE_GETENTROPY -MD -MQ lib/librte_eal.a.p/librte_eal_common
_eal_common_options.c.o -MF lib/librte_eal.a.p/librte_eal_common_eal_common_options.c.o.d
-o lib/librte_eal.a.p/librte_eal_common_eal_common_options.c.o -c ../lib/librte_eal/common
/eal_common_options.c
In file included from ../lib/librte_eal/common/eal_common_options.c:30:
In file included from ../lib/librte_eal/include/rte_memory.h:27:
In file included from ../lib/librte_eal/include/rte_fbarray.h:40:
In file included from ../lib/librte_eal/arm/include/rte_rwlock.h:12:
In file included from ../lib/librte_eal/include/generic/rte_rwlock.h:25:
In file included from ../lib/librte_eal/arm/include/rte_atomic.h:9:
```

Figure 5.5: The atomic operations error raised during the Ninja build.

The traceback of the error shown in Figure 5.5 reveals that the error comes from a header file named `rte_atomic.h`. This file defines atomic operations which are used in the DPDK framework. However, the drivers which use atomic operations should only be used for machines which have a 32-bit architecture. Therefore, they should not be compiled when using the Morello board. To prevent the incompatible drivers from compiling, they were removed from the `meson_options.txt` configuration file located in the base DPDK directory. This allowed the Ninja build to complete, which finished the compilation process.

5.3 Helloworld

To test whether the DPDK port was compiled correctly, an example application was executed. The compiled DPDK framework contains an example application called Helloworld, which was used for this purpose. This application is intended to output debug information to the terminal before printing the message “hello from core 0”. The result of running this application is shown in Figure 5.6 below.



```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/morello-build/examples]$
./dpdk-helloworld -l 0 -n 4 --no-shconf --no-huge
EAL: Sysctl reports 4 cpus
EAL: Detected 4 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Static memory layout is selected, amount of reserved memory can be adjusted with -m o
r --socket-mem
EAL: Detected static linkage of DPDK
EAL: In BSD Selected IOVA mode 'PA'
EAL: No such file or directory
EAL: No such file or directory
EAL: No such file or directory
EAL: WARNING: TSC frequency estimated roughly - clock timings may be less accurate.
EAL: No legacy callbacks, legacy socket not created
hello from core 0
```

Figure 5.6: The DPDK `helloworld` program.

The output shown in Figure 5.6 matches the expected result. Therefore, the DPDK library was successfully ported to the Morello board.

5.3.1 Usage

To run the Helloworld application, the following command is used:

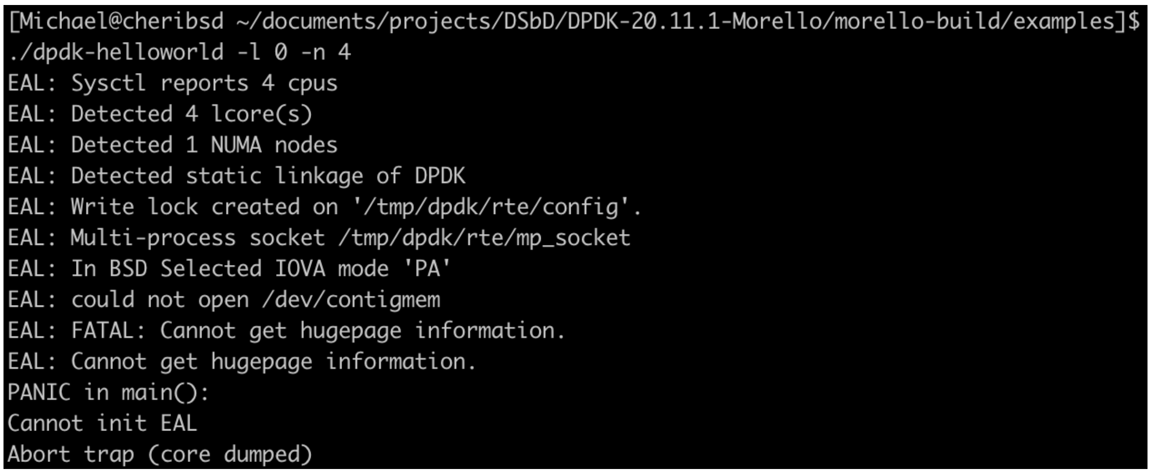
```
./dpdk-helloworld -l {core_list} -n {core_count} --no-shconf
--no-huge
```

Where:

- `-l {core_list}` is the list of cores to be used by the application [33]. This should be 0 on the Morello board.
- `-n {core_count}` is the number of cores the CPU has [33]. The Morello board has 4 CPU cores.
- `--no-shconf` disables the creation of shared files [33].
- `--no-huge` disables the use of memory hugepages [33].

5.4 Single Process Limit

The DPDK port has a limitation that means it can only use a single process. If multiple processes are used, a corresponding error is raised due to the inability to share memory between processes or use hugepages. The corresponding error is shown in Figure 5.7 below.



```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/morello-build/examples]$
./dpdk-helloworld -l 0 -n 4
EAL: Sysctl reports 4 cpus
EAL: Detected 4 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Detected static linkage of DPDK
EAL: Write lock created on '/tmp/dpdk/rte/config'.
EAL: Multi-process socket /tmp/dpdk/rte/mp_socket
EAL: In BSD Selected IOVA mode 'PA'
EAL: could not open /dev/contigmem
EAL: FATAL: Cannot get hugepage information.
EAL: Cannot get hugepage information.
PANIC in main():
Cannot init EAL
Abort trap (core dumped)
```

Figure 5.7: The DPDK error raised when using multiple processes.

As shown in Figure 5.7, the helloworld application raises an error stating that it cannot get hugepage information. To fix this error, two command line interface (CLI) arguments need to be specified, which are `--no-shconf` and `--no-huge`. These arguments limit the application to a single process [33].

The documentation for DPDK states that the FreeBSD OS is incompatible with multiple processes. To fix this limitation, the `contigmem` and `nic_uio` kernel modules must be compiled and loaded. These kernel modules allow DPDK to use contiguous memory which allows for multiple processes to be used [34]. However, CheriBSD does not currently support these modules and porting these modules was determined to be outside of the scope of this project. This is because all of the project objectives could be completed with this limitation in place.

5.5 Progress Reflection

Task 4 of the project timeline shown in Appendix A refers to the port of DPDK to the Morello board. This task was assigned 3 weeks but it took approximately a month to complete. This is because very few required changes were expected because of the architectural similarities between the Morello board and FVP. However, the time taken to make each change was greater than anticipated due to inexperience and unfamiliarity with the C programming language. In hindsight, a more conservative time estimate should have been made to reflect the uncertainty associated with porting a C library.

Objective 3 of the project specification asks for a packet processing library to be ported to the Morello board. This port of DPDK achieves that requirement. Additionally, this objective asks for CHERI capabilities to be enabled for this port. This was achieved by using the PureCap compilation mode. However, there is a limit to using a single process for each instance of an application which is built with this port. Although this limitation does not affect this project, it does provide scope for improvement.

6 Packet Processing Application

This section details the packet processing application developed for the Morello board, including how it operates and interacts with packet consumers. This relates to task 6 of the project timeline shown in Appendix A and fulfils project objective 4 from the specification.

6.1 Overview

A packet processing application, titled CHERI Networking, was created on the Morello board. This was built using the DPDK port described in Section 5 and was based on the Limelight application previously developed by Pytilia. This application was compiled as part of the DPDK compilation process, which is standard for DPDK example applications. This means the application was compiled in PureCap mode.

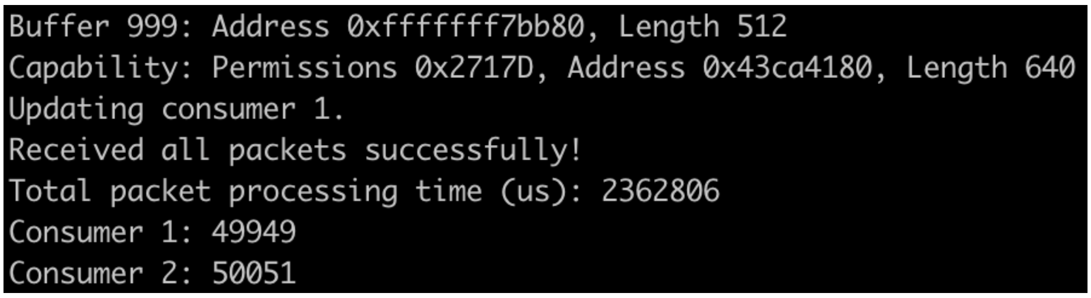
A git repository was created for this application which was pushed upstream to GitLab [35] and GitHub [36]. This repository was also added as a submodule of the git repository used for the DPDK port. This provides an example application to DPDK which uses CHERI capabilities. The git submodule also simplifies the compilation process because it ensures the application will be located in the correct subdirectory of DPDK and the required build configuration settings will be used.

CHERI Networking operates by performing the following tasks:

1. Application options are determined based on the application inputs.
2. The current time in milliseconds is stored in a local variable.
3. Up to 1,000 packets from the input packet stream are read into separate buffers.
4. Each packet is classified according to its payload data, which determines the intended consumer of the packet.
5. The packet is dispatched to the intended consumer.
6. The consumer will print the required data relating to the packet, as per the selected terminal output mode. Details on the terminal output modes are given in Section 11.
7. The consumer will increment a counter variable after receiving a packet.
8. The packet is cleared from the buffer.
9. Steps 4-8 are repeated for each of the packets stored in a buffer.
10. After all buffers have been used, the next 1,000 packets are processed by repeating steps 3-9.

11. When no more packets remain in the stream, the current time in milliseconds is stored in a local variable.
12. The total packet processing time is calculated by finding the difference between the timestamps recorded in steps 2 and 11.
13. The packet processing time and consumer counters are printed to the terminal.

Figure 6.1 below shows the output produced by CHERI Networking when run.



```
Buffer 999: Address 0xfffffff7bb80, Length 512
Capability: Permissions 0x2717D, Address 0x43ca4180, Length 640
Updating consumer 1.
Received all packets successfully!
Total packet processing time (us): 2362806
Consumer 1: 49949
Consumer 2: 50051
```

Figure 6.1: The output of CHERI Networking

As shown in Figure 6.1, CHERI Networking outputs the required information to the terminal, finishing with the total packet processing latency and consumer counters. This matches the desired behaviour.

6.2 Usage

CHERI Networking can be run from the terminal with the following command:

```
{build_dir}/dpdk-cheri_networking -l {core_list} -n {core_count}
--no-huge --no-shconf --vdev=net_pcap0,rx_pcap={stream_dir}/
{packet_stream},tx_pcap={stream_dir}/out.pcap -- {options}
```

Where:

- `build_dir` is the build directory for the compiled version of CHERI Networking. This will be the `examples` subdirectory of the Meson build directory specified when building DPDK.
- `-l {core_list}` is the list of cores to be used by the application [33]. This should be 0 on the Morello board.
- `-n {core_count}` is the number of cores the CPU has [33]. The Morello board has 4 CPU cores.

- `--no-shconf` disables the creation of shared files [33].
- `--no-huge` disables the use of memory hugepages [33].
- `--vdev` refers to the virtual device to be created [33]. This includes the name of the device and the input and output pcap files.
- `stream_dir` is the directory containing the packet streams.
- `packet_stream` is the name of the pcap file containing the desired packet stream. This name must include the file extension.
- `options` is the list of application options specified.

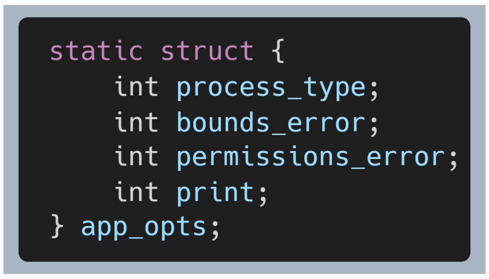
6.3 Options

CHERI Networking has various options that can be specified, which are:

- **SingleProc** - Run CHERI Networking in CHERI processing mode (uses a single process).
- **InterProc** - Run CHERI Networking in IPC processing mode.
- **NoProc** - Run CHERI Networking in no-processing mode.
- **Bounds** - Attempt to raise a bounds error by reading beyond the capability bounds.
- **Permissions** - Attempt to raise a permissions error by writing to a read-only capability.
- **Quiet** - Remove all printing to the terminal except for the consumer counters and total packet processing time.
- **Verbose** - Print all packet data and metadata to the terminal.
- **DefaultPrint** - Print the default output to the terminal.

6.3.1 Implementation

A struct is used to implement the application options in CHERI Networking, which is shown in Figure 6.2 below.



```
static struct {
    int process_type;
    int bounds_error;
    int permissions_error;
    int print;
} app_opts;
```

Figure 6.2: The struct defining the application options.

The struct shown in Figure 6.2 has 4 attributes. However, there are 8 application options listed. This is because some options are mutually exclusive and cannot be used together. Therefore, these contradictory options use different values for the same attribute. The mappings between the options and struct fields are shown in Table 6.1 below.

Option	Struct Member	Value
SingleProc	`process_type`	1
InterProc	`process_type`	2
NoProc	`process_type`	3
Bounds	`bounds_error`	1
Permissions	`permissions_error`	1
Quiet	`print`	0
Verbose	`print`	2
DefaultPrint	`print`	1

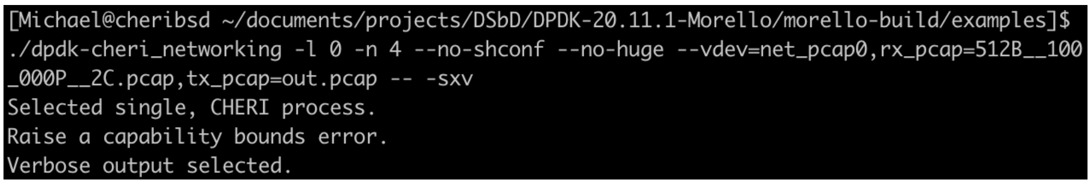
Table 6.1: Options for CHERI Networking mapped to struct attributes.

A process type must be specified otherwise an error will be raised by the application. Multiple process types can be specified, but only one will be used. The priority is given to no-processing, followed by IPC, followed by CHERI, which has the lowest priority. More detail on the processing modes can be found in Section 6.4.

A bounds or permissions error may be specified for any processing type without raising an error. If both are specified, then the bounds error will be raised first. However, these options only have functionality when using the CHERI processing mode. This is because CHERI capabilities are only required for validation when using this processing type. More detail on capability validation can be found in Section 7.

Additionally, specifying both quiet and verbose printing will result in verbose mode being prioritised. Specifying neither will result in the default terminal output being used. More detail on terminal output can be found in Section 11.

The selected options will be printed to the terminal at the beginning of the application execution, unless quiet mode is selected. The output given when the SingleProc, Verbose, and Bounds options are selected is shown in Figure 6.3 below.



```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/morello-build/examples]$
./dpdk-cheri_networking -l 0 -n 4 --no-shconf --no-huge --vdev=net_pcap0,rx_pcap=512B__100
_000P__2C.pcap,tx_pcap=out.pcap -- -sxv
Selected single, CHERI process.
Raise a capability bounds error.
Verbose output selected.
```

Figure 6.3: CHERI Networking printing the selected application options.

Figure 6.3 shows that all application options are correctly displayed to the terminal by CHERI Networking. This matches the expected behaviour.

6.3.2 Usage

The application options can be specified by either a CLI argument or environment variable. Environment variables are globally defined within the shell environment and do not expire, so these are used to specify a default behaviour desired for the application. To use an environment variable, it should be assigned a value of "yes" in the shell environment. This matches the behaviour used in the Limelight application and can be achieved through the `export` command. CLI arguments are specified during the invocation of the application and are only applicable to that application instance. For this reason, CLI arguments take precedence over environment variables when contradicting options are specified, even if the CLI argument grants an option with a lower priority than the environment variable. The CLI arguments and environment variables are shown in Table 6.2 below.

Option	CLI Argument	Environment Variable
SingleProc	`-s`	`PYTILIA_SINGLE_PROCESS`
InterProc	`-i`	`PYTILIA_INTER_PROCESS`
NoProc	`-n`	`PYTILIA_NO_PROCESS`
Bounds	`-x`	`PYTILIA_BOUNDS_ERROR`
Permissions	`-y`	`PYTILIA_PERMISSIONS_ERROR`
Quiet	`-q`	`PYTILIA_QUIET`
Verbose	`-v`	`PYTILIA_VERBOSE`
DefaultPrint	None	None

Table 6.2: Usage of CHERI Networking.

6.3.3 Testing and Validation

To validate that the application options can be specified correctly, all CLI arguments and environment variables were tested individually. When testing options that did not relate to the processing type, CHERI processing mode was used. These tests all worked as expected, validating that the implementation was correct. Additional unit tests were performed to determine how different options interacted with each other. A table detailing the additional unit tests is shown in Appendix B.

6.4 Processing Modes

This section details the different processing modes which CHERI Networking can use when dispatching packets to a consumer plugin.

6.4.1 Overview

CHERI Networking can dispatch packets in the following different methods:

- **CHERI processing mode** - This involves dispatching packets to consumers via CHERI capabilities.
- **IPC processing mode** - This involves dispatching packets to consumers via User Datagram Protocol (UDP) sockets.
- **No processing mode** - This involves reading packets into a buffer and not dispatching them to any consumers.

A diagram depicting the different communication methods between CHERI Networking and the packet consumers is shown in Figure 6.4 below.

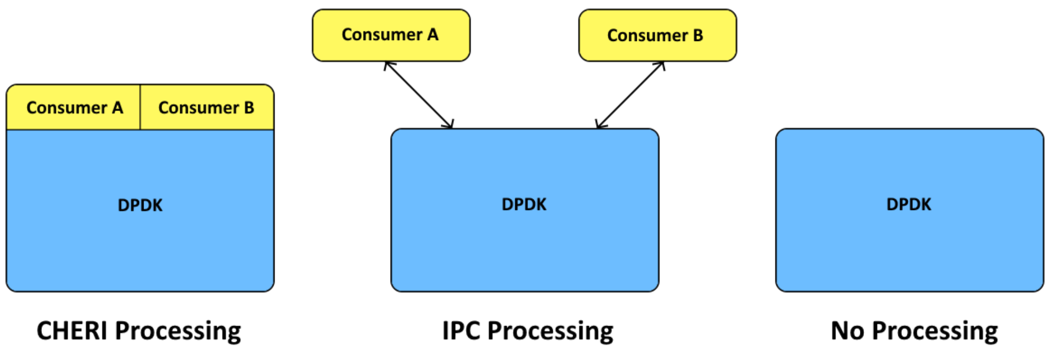


Figure 6.4: Comparison of the communications between CHERI Networking and consumers.

6.4.2 CHERI

When running CHERI Networking in CHERI processing mode, the application and the consumer plugin are run in the same process. This means that they all share the same virtual address space. This allows virtual addresses which are valid for the main application to be used in the plugins.

CHERI Networking will dispatch packets to a consumer by passing a CHERI capability associated with the original packet data. This capability is validated for security reasons, which is discussed in more detail in Section 7 of this report.

CHERI Networking can be run in CHERI processing mode by a CLI argument, `-s`, or an environment variable, `PYTILIA_SINGLE_PROCESS`.

6.4.3 IPC

When running CHERI Networking in IPC processing mode, the application and the consumer plugin are run in separate processes. This means that they have different virtual address spaces. Therefore, a virtual address cannot be passed to the consumer because it will not map to a meaningful address in physical memory.

CHERI Networking will dispatch packets to a consumer through the use of UDP sockets. This was chosen because UDP is a method of IPC which is known for its high speed, both in terms of latency and throughput [37]. UDP involves sending a copy of the packet data to the consumer through UDP sockets. This means that both CHERI Networking and the packet consumer will only have access to their respective copies of the packet data. Therefore, the security required for the CHERI processing mode is not required when using IPC because the consumer cannot access the original packet data.

To use IPC processing mode, another application must run concurrently with CHERI Networking to receive the packets. This packet receiver application contains the packet consumer and is run in a separate process from the CHERI Networking application. Details on this application can be found in Section 8.

The UDP sockets created to facilitate communications between CHERI Networking and the packet receiver operate across localhost. This is possible because both applications run on the same machine at the same time. The use of localhost was chosen because it does not use any external network, which means that the data is not transferred through any external medium. This increases the speed of communication.

The localhost ports used when creating the UDP sockets are determined by adding the recipient consumer number to 5000. Port 5000 was chosen as the lowest port because it is standard for use in localhost due to its availability and ease of use [38]. For example, consumer 1 will use port 5001. The UDP sockets are created after the application options are determined and before the first time reading is recorded. This is because the time taken to create the sockets should not be included when calculating the packet processing latency.

Additionally, after all of the packets are dispatched to the consumers, CHERI Networking will listen on a UDP socket. This is because the application is awaiting a packet from each of the consumers containing their final counter value. This occurs after the final time reading has been recorded. This ensures the extra packet transmissions are not included in the total packet processing time to keep a fair comparison with the CHERI processing mode.

CHERI networking can be run in IPC processing mode by a CLI argument, `-i`, or an environment variable, `PYTILIA_INTER_PROCESS`.

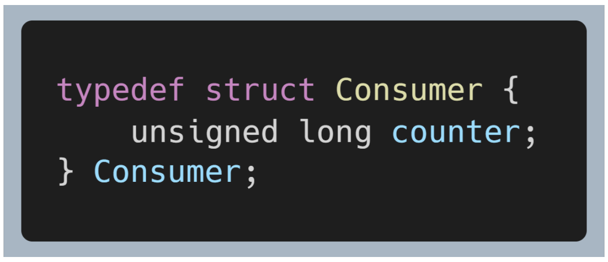
6.4.4 No-Processing

When running CHERI Networking in CHERI no-processing mode, the application does not interact with any consumer plugins. Each packet is read into a buffer, matching the behaviour of the other processing modes. However, this packet is then discarded before the next packet is read into the buffer. All other behaviour remains unchanged from the other modes of operation. This means that the time taken to start up the application, read packets into the buffers, and terminate the application is constant between all 3 processing modes. Therefore, this processing mode provides a valid control to compare the other processing modes with when evaluating their performance.

CHERI Networking can be run in no-processing mode by a CLI argument, `-n`, or an environment variable, `PYTILIA_NO_PROCESS`.

6.5 Packet Consumers

The packet consumer and its related functions are defined in a separate C header and source file to the main packet processing application. This makes the code portable so that it can easily be reused in both the packet processing application and the packet receiver application. The struct which defines the consumer is shown in Figure 6.5 below.



```c
typedef struct Consumer {
    unsigned long counter;
} Consumer;
```

Figure 6.5: The struct defining the packet consumer.

As shown in Figure 6.5, the consumer contains only one attribute, which is the counter variable. This is because the consumer is designed to be simple for the purposes of this project as the focus is on the dispatch of packets to the consumers. However, the use of a struct instead of a standalone variable makes the consumer more expandable for future work.

In addition to the struct, the following functions are defined in the consumer header/source files:

- `consumer_increment_counter` - This function is used to increment the counter variable of the consumer by 1. This is called every time a consumer receives a packet.
- `consumer_print_details` - This function is used to print the consumer counter variable to the terminal.

6.6 Progress Reflection

Task 6 of the project timeline shown in Appendix A covers the creation of the packet processing application. This task was allocated 8 weeks, not including the 2-week break over the Christmas holiday period. However, this work was completed in approximately 5 weeks, including the development of the no-processing operation mode which was not planned in the timeline. This is because significantly more time was committed to the project during these weeks to ensure the project was ahead of schedule before coursework for other modules was going to be released.

Objective 4 of the project specification covers the creation of a packet processing application. This was achieved in this section, excluding the packet receiver detailed in Section 8. CHERI Networking contains both CHERI and IPC processing modes to interact with packet consumers, as per the specification. However, the no-processing mode was not required. This was developed to improve the performance evaluation required by objective 7.

7 Capability Validation

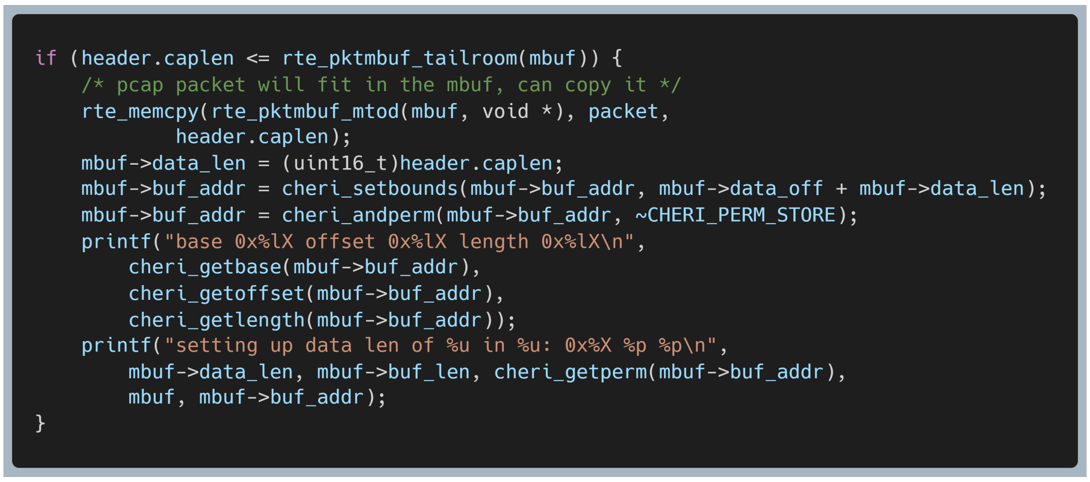
This section covers the validation of the CHERI capabilities used in the packet processing application when run in CHERI processing mode. This relates to task 6.2 of the project timeline shown in Appendix A and fulfils project objective 5 from the specification.

7.1 Overview

When running the packet processing application in the CHERI processing mode, the consumer plugins are run in the same process and address space as the application. This means that the virtual addresses sent to the plugins can be dereferenced, allowing the consumer to view the data stored at that address. Therefore, if an unprotected pointer was sent to a consumer plugin, a security risk would be introduced relating to unauthorised access to data. This is because the consumer could access data beyond the packet length by using pointer arithmetic. Furthermore, the consumer would have no restrictions on permissions, meaning that the original packet data could be overwritten. However, if a CHERI capability is sent, then access to the memory is controlled by the source application rather than the plugin. This is accomplished by defining the capability bounds and permissions to safely mitigate the aforementioned issues.

7.2 Derived Capabilities

In the previous Limelight application made by Pytilia, the CHERI capabilities associated with the packets had their bounds and permissions secured through a modification to the DPDK framework. The modified code for the DPDK function `eth_pcap_rx` is shown in Figure 7.1 below.

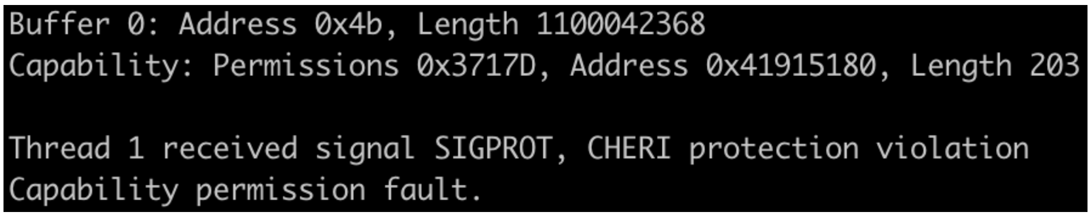


```
if (header.caplen <= rte_pktmbuf_tailroom(mbuf)) {
    /* pcap packet will fit in the mbuf, can copy it */
    rte_memcpy(rte_pktmbuf_mtod(mbuf, void *), packet,
            header.caplen);
    mbuf->data_len = (uint16_t)header.caplen;
    mbuf->buf_addr = cheri_setbounds(mbuf->buf_addr, mbuf->data_off + mbuf->data_len);
    mbuf->buf_addr = cheri_andperm(mbuf->buf_addr, ~CHERI_PERM_STORE);
    printf("base 0x%lX offset 0x%lX length 0x%lX\n",
        cheri_getbase(mbuf->buf_addr),
        cheri_getoffset(mbuf->buf_addr),
        cheri_getlength(mbuf->buf_addr));
    printf("setting up data len of %u in %u: 0x%X %p %p\n",
        mbuf->data_len, mbuf->buf_len, cheri_getperm(mbuf->buf_addr),
        mbuf, mbuf->buf_addr);
}
```

Figure 7.1: Source code snippet from the DPDK function `eth_pcap_rx`.

As shown in Figure 7.1, the `eth_pcap_rx` function was modified to restrict the capabilities whenever a packet was read from the input packet stream. This was achieved through the `cheri_setbounds` and `cheri_setperms` functions. This restricted capability was returned to the Limelight application, which used it as a packet buffer variable. This resulted in a buffer automatically having its bounds and permissions restricted whenever a new packet was received. This means that the Limelight application could not enlarge the bounds or add permissions due to the monotonicity feature of CHERI capabilities.

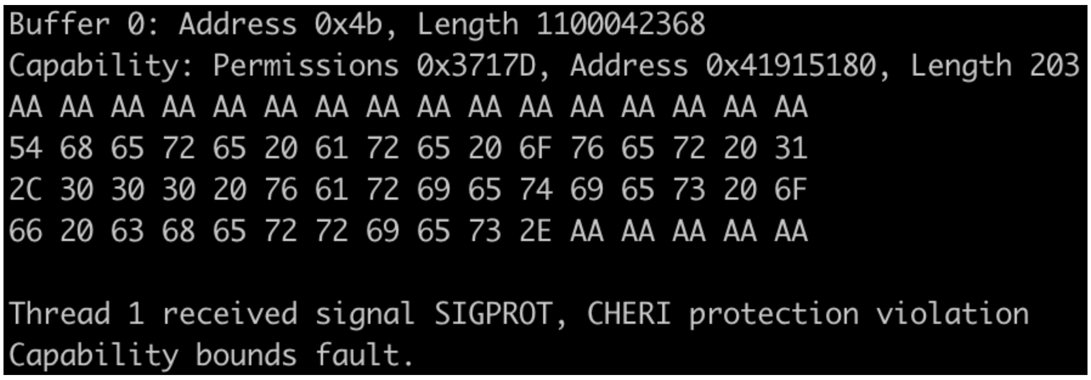
This implementation caused a problem whenever a buffer was reused to store a new packet. This would be required when the packet stream contained more packets than the number of buffers in the application, which was 16. This is because the application would attempt to write new packet data to a read-only buffer. This raises a capability permissions fault, as shown in Figure 7.2 below.



```
Buffer 0: Address 0x4b, Length 1100042368
Capability: Permissions 0x3717D, Address 0x41915180, Length 203

Thread 1 received signal SIGPROT, CHERI protection violation
Capability permission fault.
```

Figure 7.2: Attempting to reuse a buffer in Limelight.

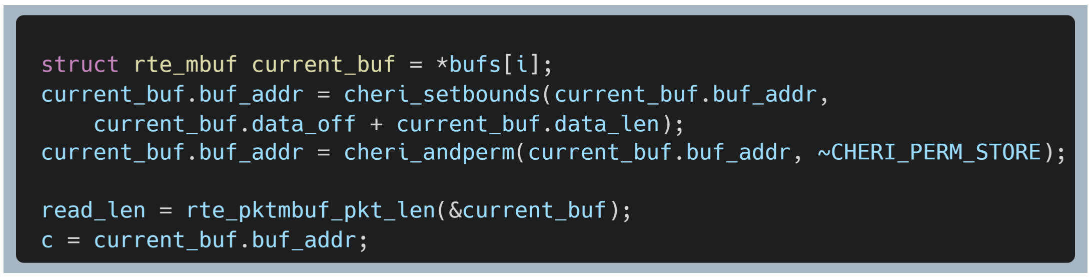
Removing the permissions validation to allow write access to the buffer allows Limelight to reuse the packet buffer. However, this implementation remained unreliable when reading packet streams with more than 16 packets. This is because, if a larger packet was read into the buffer, the capability bounds would be too short to allow the entire packet data to be stored. This raises a capability bounds fault, as shown in Figure 7.3 below.



```
Buffer 0: Address 0x4b, Length 1100042368
Capability: Permissions 0x3717D, Address 0x41915180, Length 203
AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA
54 68 65 72 65 20 61 72 65 20 6F 76 65 72 20 31
2C 30 30 30 20 76 61 72 69 65 74 69 65 73 20 6F
66 20 63 68 65 72 72 69 65 73 2E AA AA AA AA AA

Thread 1 received signal SIGPROT, CHERI protection violation
Capability bounds fault.
```

Figure 7.3: Attempting to reuse a buffer for a larger packet in Limelight.

To solve this issue, the capability validation was removed from the `eth_pcap_rx` function. Instead, CHERI Networking derives a new capability from the packet buffer and stores this in a local struct called `current_buf`. This struct contains an address which points to the original packet data stored in the buffer. The corresponding code for this is shown in Figure 7.4 below.



```
struct rte_mbuf current_buf = *bufs[i];
current_buf.buf_addr = cheri_setbounds(current_buf.buf_addr,
    current_buf.data_off + current_buf.data_len);
current_buf.buf_addr = cheri_andperm(current_buf.buf_addr, ~CHERI_PERM_STORE);

read_len = rte_pktmbuf_pkt_len(&current_buf);
c = current_buf.buf_addr;
```

Figure 7.4: Source code snippet from CHERI Networking.

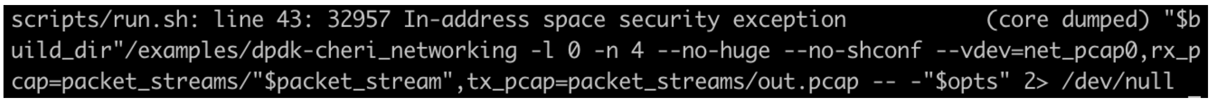
The source code from Figure 7.4 secures the newly derived CHERI capability in the CHERI Networking application instead of through the DPDK framework. This allows a buffer to be reused by a larger packet. This capability is secured by having its bounds tightened and permissions restricted to maintain the previous level of security.

Only the untrusted, third-party consumer plugins must have their access to the original packets restricted, not the networking application itself. Therefore, security is maintained by passing this derived capability to the consumer. Furthermore, this gives the application write access to the original packet, which was not previously permitted. One example where this may be useful in industry is if the application is required to write an extension header for an IPv6 packet [39].

7.3 Bounds Validation

The capability bounds are used to ensure that a consumer cannot access memory outside of the received packet data. This is achieved by programmatically shrinking the bounds so that they range from exactly the first to the last byte of packet data.

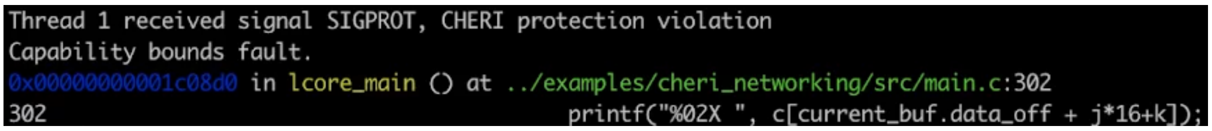
The packet processing application can be run such that it attempts to read one character beyond the bounds of the CHERI capability it receives. This can be performed by either a CLI argument, `-x`, or an environment variable, `PYTILIA_BOUNDS_ERROR`. The corresponding application output can be seen in Figure 7.5 below.



```
scripts/run.sh: line 43: 32957 In-address space security exception          (core dumped) "$b
uild_dir"/examples/dpdk-cheri_networking -l 0 -n 4 --no-huge --no-shconf --vdev=net_pcap0,rx_p
cap=packet_streams/"$packet_stream",tx_pcap=packet_streams/out.pcap -- -"$opts" 2> /dev/null
```

Figure 7.5: Attempting to read beyond capability bounds.

Figure 7.5 shows that the packet processing application raises an in-address space security exception when trying to read beyond the capability bounds. This shows that the consumer cannot access the next byte of data. However, this error message does not specify the capability bounds caused the exception to be raised. To examine this error in more detail, the application was run using the CHERI GDB debugger. The corresponding output is shown in Figure 7.6 below.



```
Thread 1 received signal SIGPROT, CHERI protection violation
Capability bounds fault.
0x000000000001c08d0 in lcore_main () at ../examples/cheri_networking/src/main.c:302
302                                     printf("%02X ", c[current_buf.data_off + j*16+k]);
```

Figure 7.6: Attempting to read beyond capability bounds using CHERI GDB.

As shown in Figure 7.6, the application raises a capability bounds fault, which is the desired outcome. This proves that the capability bounds are restricted correctly to match the length of the packet data.

To ensure that the capability bounds are correctly applied for different packet lengths, a variety of packet streams were tested with the application. All of the test packet streams from Section 9.2 were used because they contained different packet lengths. These all yielded the same results.

7.4 Permissions Validation

The capability permissions are used to ensure that a consumer can only read the data stored within the bounds of the CHERI capability. This is achieved by programmatically removing the write permission. It is also important to note that the capability should not have execute permissions. However, this permission is not granted to the derived capability upon creation so it does not need to be removed. Therefore, only the read permission remains for the capability.

The packet processing application can be run such that it attempts to overwrite the first byte of the packet data associated with a read-only capability. This can be performed by either a CLI argument, `-y`, or an environment variable, `PYTILIA_PERMISSIONS_ERROR`. The corresponding application output can be seen in Figure 7.7 below.

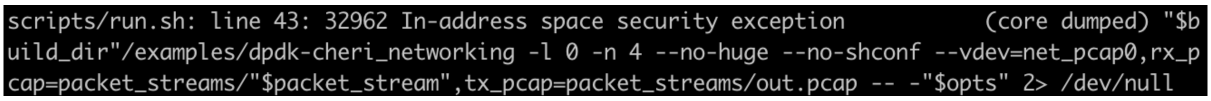

```
scripts/run.sh: line 43: 32962 In-address space security exception         (core dumped) "$b
uild_dir"/examples/dpdk-cheri_networking -l 0 -n 4 --no-huge --no-shconf --vdev=net_pcap0,rx_p
cap=packet_streams/"$packet_stream",tx_pcap=packet_streams/out.pcap -- -"$opts" 2> /dev/null
```

Figure 7.7: Attempting to overwrite read-only packet data.

Figure 7.7 shows that the packet processing application raises an in-address space security exception when trying to overwrite the packet data. This shows that the consumer does not have write permissions. However, this error message does not specify that the capability permissions caused the exception to be raised. To examine this error in more detail, the application was run using the CHERI GDB debugger. The corresponding output is shown in Figure 7.8 below.

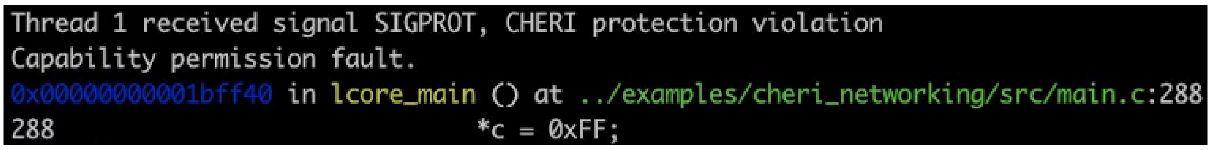

```
Thread 1 received signal SIGPROT, CHERI protection violation
Capability permission fault.
0x000000000001bff40 in lcore_main () at ../examples/cheri_networking/src/main.c:288
288                         *c = 0xFF;
```

Figure 7.8: Attempting to overwrite read-only packet data using CHERI GDB.

As shown in Figure 7.8, the application raises a capability permission fault, which is the desired outcome. This proves that a consumer has read-only access rights and that the capability permissions are correctly restricted.

To ensure that the capability permissions are correctly applied for different packets, all the test packet streams from Section 9.2 were used to test the application. These all yielded the same results.

7.5 Progress Reflection

Task 6.2 of the project timeline shown in Appendix A refers to adding capability validation to the packet processing application. A week was allocated to this task because the expectation was to implement capability validation using the same methodology that was used in the previous Limelight application. The previous method was implemented in two days because the DPDK port to the Morello board was forked from the port to the Morello FVP, so the implementation required very few changes. However, as discussed, it was discovered that this method was insufficient for this project. This was not known before this project began because the largest packet stream used for testing on the Limelight application contained two packets, so no buffers were ever reused. Therefore, this could not have been planned for in advance. Implementing derived capabilities required another week. This meant the overall time spent on this task was slightly longer than planned.

Objective 5 of the project specification asks for capability bounds and permissions to be validated. The expectation was to validate the application in the same way as the previous application. Therefore, the use of derived capabilities has added value to this objective as it allows buffers to be reused, removing an upper limit to the packet count. It also allows an application to retain full control over the CHERI capabilities instead of inheriting already restricted capabilities from the DPDK framework.

8 Packet Receiver

This section details the packet receiver developed to interact with CHERI Networking when using the IPC processing mode. This relates to task 6.5 of the project timeline shown in Appendix A and fulfils part of project objective 4 from the specification.

8.1 Overview

A packet receiver application was created to receive the packets sent by CHERI Networking when using the IPC processing method. This application contains a packet consumer which matches the consumer definition from Section 6.5. This means that the application has its own virtual address space and cannot access the virtual memory associated with CHERI Networking. This application uses UDP sockets to communicate over localhost, which corresponds with the IPC communication method used in CHERI Networking.

This application was compiled in hybrid mode without the use of any CHERI capabilities. This is because the packet receiver is used exclusively with the IPC processing mode of CHERI Networking, so there is no requirement to use CHERI capabilities.

The packet receiver application was added to a git repository which was pushed upstream to GitLab [40] and GitHub [41].

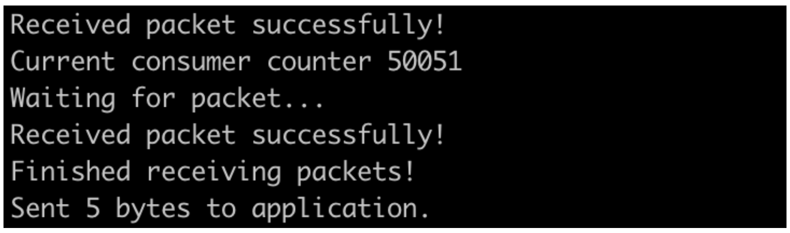
The packet receiver operates by performing the following tasks:

1. A UDP socket is created with a port which corresponds to the consumer number.
2. The application listens on localhost using the specified port until a packet is received.
3. When a packet is received, the application stores it in a buffer.
4. The consumer will increment its counter variable.
5. The application will print the required data relating to the packet, as per the selected terminal output mode. Details on the terminal output modes are given in Section 11.
6. Steps 2-5 are repeated until a packet which stores the string “FINISHED” is received.
7. The application prints to the terminal that it has finished receiving packets. This step is skipped if using quiet terminal output.
8. The application sends a UDP packet containing the final consumer counter to CHERI Networking via localhost.

9. The number of bytes sent to CHERI Networking is printed to the terminal. This step is skipped if using quiet terminal output.
10. The application terminates.

The localhost port used when creating the UDP socket is determined by the same method as used in CHERI Networking, which is detailed in Section 6.4.3.

Figure 8.1 below shows the output produced by the packet receiver when finished receiving packets.



```
Received packet successfully!
Current consumer counter 50051
Waiting for packet...
Received packet successfully!
Finished receiving packets!
Sent 5 bytes to application.
```

Figure 8.1: The output of the packet receiver application.

As shown in Figure 8.1, the packet receiver prints the required information to the terminal. It ends with a message stating that it has finished receiving packets and the number of bytes it has sent to CHERI Networking. This matches the desired behaviour.

8.2 Usage

The packet receiver is compiled with the `make` command. This is because a corresponding Makefile was created for the application.

To run the packet receiver, the following command is used in the terminal:

```
./application.o {consumer_number} {print_mode}
```

Where:

- `consumer_number` is the number associated with the consumer.
- `print_mode` is the terminal output mode to be used. This is an optional argument.

To communicate with CHERI Networking, the consumer number should be either a 0 or 1 as these are the indexes used internally for each of the 2 consumers. Further information on the terminal output modes can be found in Section 11.

The packet receiver must begin execution before CHERI Networking. This ensures that the application will be ready to receive the first packet that CHERI Networking dispatches so that none of the packets will be missed. All consumers are run in separate instances of the packet receiver application. Therefore, to create two consumers, two instances of the packet receiver should be run concurrently. These instances may be run in separate terminal sessions or the background of the same terminal session by using the `&` operator.

8.3 Progress Reflection

Task 6.5 of the project timeline shown in Appendix A refers to the creation of a packet receiver application. 1 week was allocated to this task which was an accurate estimation. This allowed time for the application to be created and communicate with CHERI Networking via UDP sockets. Furthermore, Task 6.8 refers to the implementation of the packet consumer to the packet processing application, which was also allocated one week. This was completed in 2 days of work. This is because the consumer plugin was created for CHERI Networking in dedicated C header/source files which were portable and easy to integrate with the design of the packet receiver application.

This application complements the IPC processing mode of CHERI Networking, which completes part of objective 4 of the project specification.

9 Packet Generation

This section covers the generation of different packet streams for use with the packet processing application. This relates to task 5 of the project timeline shown in Appendix A and fulfils project objective 6 from the specification.

9.1 Overview

A Python script was created to generate a variety of packet streams and store them in a packet capture (pcap) file. Raw Ethernet packets were generated for these packet streams, meaning only the packet payloads were defined. Each packet generated contained a payload beginning with a Unicode character in the decimal range of 160-169. These characters were chosen because they do not commonly occur in data strings which made the classification of packets very simple. This also allowed for the classification of up to ten packet consumers. This script uses the `python-libpcap` library to handle pcap files. The generated pcap files were used as the input network traffic for CHERI Networking.

The Python script also contains a function, `calculate_consumer_count`, which calculates the expected counters for each packet consumer based on the packet stream data. This function reads the contents of a specified pcap file and uses the same classification method as CHERI Networking to determine the expected counter values.

A git repository was created for this script and was pushed upstream to GitLab [42] and GitHub [43]. The generated packet streams could not be added to this git repository because some of the streams exceeded the GitHub file size limit of 100 MB [44]. Therefore, only the Python scripts were added to the git repository.

9.2 Test Packet Streams

Initially, four packet streams were created to allow for testing of the packet processing application. Details of these streams are shown in Table 9.1 below.

Stream	Packet Count	Packet Length (Bytes)
1	6	75 - 131
2	10,000	75 - 131
3	20,000	75
4	20,000	75 - 131

Table 9.1: Details of the test packet streams.

As can be seen from Table 9.1, stream 1 contains 6 packets which vary in length. This allowed the packet processing application to be tested quickly. It was also the only stream which could be used initially when the application had a limit of 16 packets, which was a limitation inherited from the Limelight application developed by Pytilia. Stream 2 contains 10,000 packets, also of varying lengths. This is the maximum number of packets which could be reliably read by the application when it used 10,000 buffers for storing packets. Stream 3 contains 20,000 packets of a constant length. This stream could be successfully read by the application whenever it had 10,000 buffers because none of the buffers read a new packet which was larger than the previous one. Stream 4 contains 20,000 packets with varying lengths, meaning that the buffers in the application had to be reused. This stream proved that there was a capability validation issue with the previous packet processing application made by Pytilia, which had to be fixed for this project.

9.3 Benchmarking Packet Streams

To enable performance metrics to be recorded for the packet processing application, packet streams were created which varied in terms of packet count and packet size. Each packet in these streams contained a payload beginning with a Unicode character with a decimal representation of either 160 or 161. This is because the final version of CHERI Networking interacted with 2 consumers and these were the relevant characters. The character was chosen randomly for each packet in the stream, with equal probability weightings for each character at 50%. Furthermore, the expected consumer counters were calculated through the `calculate_consumer_count` function of the packet generator script. The consumer counters calculated are specific to the streams generated for use in this project. The random nature of the packet generation means that if the same script is used to generate new packet streams, the expected counter values may vary.

9.3.1 Packet Size

The packet streams with a variable packet size are detailed in Table 9.2 below.

Packet Count	Packet Length (Bytes)	Expected Counter 0	Expected Counter 1
100,000	128	49,998	50,002
100,000	258	50,040	49,960
100,000	512	50,146	49,854
100,000	1,024	49,957	50,043
100,000	2,048	49,781	50,219
100,000	4,096	50,013	49,987
100,000	8,192	50,200	49,800

Table 9.2: Packet streams with variable packet sizes.

As can be seen from Table 9.2, multiple packet streams which vary in packet length have been created. After discussion with the project supervisor, it was decided that it would be best to use packets which have a length of a power of 2 bytes because these sizes are standard for benchmarking networking applications. Therefore, the packets range from 128B to 8,192B in length, with each stream containing packets which are double the length compared to the previous stream. The packet count is controlled at a constant value of 100,000 because this ensures that the performance tests could be completed in a reasonable length of time, while also providing a large enough set of test data to record accurate performance metrics.

9.3.2 Packet Count

The packet streams with a variable packet count are detailed in Table 9.3 below.

Packet count	Packet length (Bytes)	Expected Counter 0	Expected Counter 1
20,000	512	10,014	9,986
40,000	512	20,042	19,958
60,000	512	30,056	29,944
80,000	512	40,095	39,905
100,000	512	50,146	49,854
120,000	512	60,223	59,777
140,000	512	70,303	69,697
160,000	512	79,671	80,329
180,000	512	89,784	90,216
200,000	512	99,749	100,251

Table 9.3: Packet streams with variable packet counts.

Additionally, Table 9.3 shows that multiple packet streams have been generated which vary in packet count. The packet count ranges from 20,000 to 200,000 in increments of 20,000 packets. This gives a sufficient range to observe if packet count has any effect on the performance of the packet processing application. The packet size is controlled at a constant value of 512B because this is the smallest packet size which can reliably be used with the packet processing application when running in quiet mode. Therefore, the time required to record the required performance metrics is kept to a minimum. Out of the packet sizes generated, 512B is also the closest to the "average" internet packet size of between approximately 300B and 400B [45].

9.3.3 Testing and Validation

To validate that CHERI Networking processed all packets correctly, the expected consumer counters for all benchmarking packet streams were manually compared to the results produced in both the CHERI and IPC processing modes using quiet terminal output. The tables comparing the expected values to the actual values are included in Appendix C. It was shown that all packet streams were processed successfully when using CHERI processing mode. However, the packet streams containing packets of size 128B and 256B were not processed correctly when using the IPC processing mode because their consumer counters were lower than expected.

The low consumer counts occur because smaller packets require less time to transmit through UDP sockets, so more packets are transmitted in the same amount of time. Therefore, the packet receiver will receive packets at a faster rate than it can process. Additionally, UDP sockets do not establish a connection between the source and destination, so no acknowledgement is sent to confirm that a packet has been received successfully. Therefore, packets that are lost do not get retransmitted [46]. This issue could be fixed if Transmission Control Protocol (TCP) sockets were used instead of UDP sockets. This is because TCP uses acknowledgement frames to confirm if a packet has been received successfully and retransmits packets which were not successful [47].

9.4 Progress Reflection

Task 5 of the project timeline shown in Appendix A covers the generation of packet streams. One week was allocated to the generation of packet streams, which was sufficient for creating the Python script and initial test streams discussed in Section 9.2. However, the streams created for benchmarking were not accounted for in the allotted time. These streams were generated after CHERI Networking was validated to be working as intended and did not appear in the planned timeline. This took two days of work because the script to generate the streams was already created, so a large portion of the work was completed. However, some minor refactoring took place to make it easier to create this range of packet streams. Therefore, the timeline could have been improved with the inclusion of a separate task to generate packet streams for benchmarking. Additionally, the tests conducted to compare the expected consumer counts to the actual consumer counts were not considered during project development. The development process revealed the issue with using small packets with the IPC processing mode of CHERI Networking, but formal tests were not considered until a relevant question was asked during the oral examination. Completing these tests required another day of work. In hindsight, these tests should have been planned for in advance and conducted at a much earlier time.

Objective 6 of the project specification only asks for the generation of packet streams with a range of packet sizes. Therefore, the creation of packet streams with varying packet counts goes beyond this specification objective.

10 Run Script

This section details a run script which was created for CHERI Networking. This was unplanned work so it does not appear on the project timeline shown in Appendix A and adds value to the specification.

10.1 Overview

A Bash script was created to simplify the usage of CHERI Networking. This script, named `run.sh`, is located in the `scripts` subdirectory of the CHERI Networking git repository on GitHub [48] and GitLab [49].

This script was not required by the project objectives, but it was created to accomplish the following goals:

- Simplify the terminal input required to run CHERI Networking.
- Allow for CHERI Networking to be easily used outside of the DPDK build directory.
- Allow for validation that a packet stream exists.

The CHERI Networking application is designed specifically to run on a Morello board using CheriBSD. Therefore, the CLI arguments which are specific to the architecture could be hard-coded into the run script, which are the core list and the CPU core count. Additionally, the CLI arguments which are constant for the usage of CHERI Networking could also be hard-coded, which are `--no-huge` and `--no-shconf`. This simplifies the usage of the application.

The run script is located within the CHERI Networking git repository, which is a submodule of the DPDK repository. Therefore, relative paths to directories within DPDK could be hard-coded into the script. This allowed for the CHERI Networking directory to be used to run the application. This means that the script can easily switch between different builds when required. This is useful when multiple DPDK builds have been generated, such as when testing new code, and is more user-friendly than switching between directories within DPDK.

The run script also checks if the input packet stream exists. This could not be done in CHERI Networking directly due to limitations in the DPDK framework. Therefore, using a packet stream which does not exist caused the application to hang indefinitely. This script assumes the packet stream is sourced from the `packet_streams` subdirectory of the CHERI Networking repository so that only the name of the packet stream must be specified. This also simplifies the terminal input required for using the application.

The run script also allows the application options detailed in Section 6.3 to be specified.

10.2 Usage

This script is run by the following terminal command:

```
./run.sh {build_dir_name} {packet_stream} {options}
```

Where:

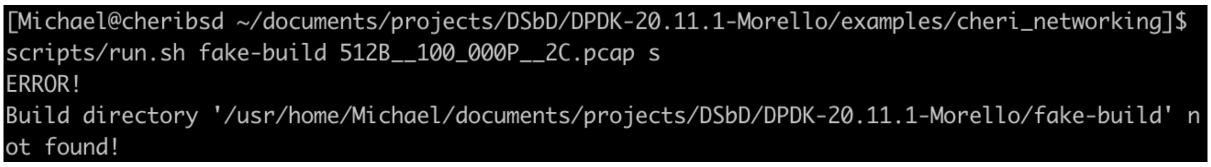
- `build_dir_name` is the name of the Meson build directory containing the compiled version of CHERI Networking. This is only the directory name, not a path to the directory.
- `packet_stream` is the name of the pcap file containing the desired packet stream. This name must include the file extension.
- `options` is the list of application options specified. This is a single string containing all of the required CLI arguments without any hyphens.

10.3 Testing and Validation

The script contains validation for all of the inputs. If an erroneous input is detected, a meaningful error is displayed on the terminal.

10.3.1 Build Directory

The `build_dir_name` is validated to check if it exists and contains the compiled CHERI Networking application. The directory will not exist if a Meson build has not been completed. If this is the case, then the run script is expected to detect this and raise an appropriate error stating that the directory is not found. Figure 10.1 below shows the response generated when a non-existent build directory is used in the run script.

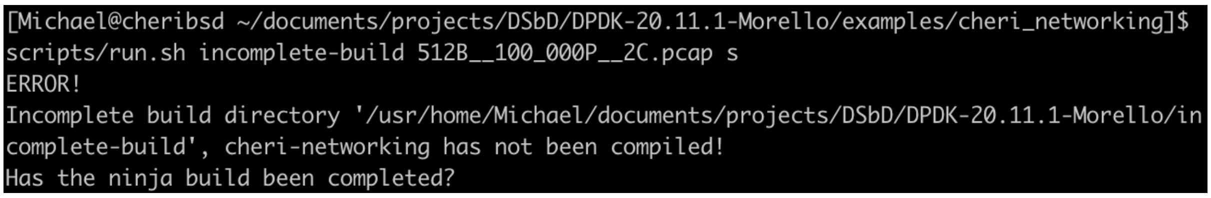


```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/examples/cheri_networking]$
scripts/run.sh fake-build 512B__100_000P__2C.pcap s
ERROR!
Build directory '/usr/home/Michael/documents/projects/DSbD/DPDK-20.11.1-Morello/fake-build' n
ot found!
```

Figure 10.1: A missing build directory caught by the run script.

Figure 10.1 shows that the missing build directory is successfully detected by the run script and an error explaining the directory was not found is raised. This matches the expected behaviour.

Similarly, the application will not be compiled if the Ninja build has not been completed, even if the Meson build has been completed. Therefore, the run script should detect if a build directory Meson build directory exists but does not contain the compiled CHERI Networking application. This should result in an error being raised which states that the build directory is incomplete. Figure 10.2 below shows the response generated when an incomplete build directory is used in the run script.



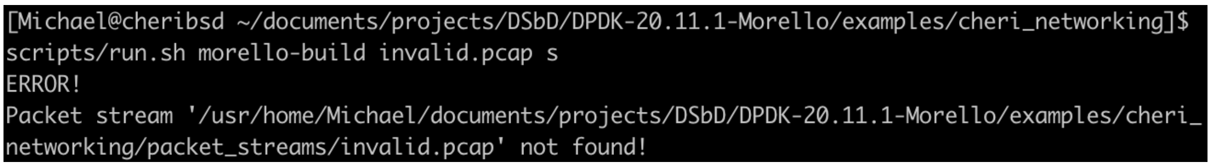
```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/examples/cheri_networking]$
scripts/run.sh incomplete-build 512B__100_000P__2C.pcap s
ERROR!
Incomplete build directory '/usr/home/Michael/documents/projects/DSbD/DPDK-20.11.1-Morello/in
complete-build', cheri-networking has not been compiled!
Has the ninja build been completed?
```

Figure 10.2: An incomplete build directory caught by the run script.

Figure 10.1 shows that the invalid build directory is successfully detected by the run script and an error explaining that the build is incomplete is raised. This matches the expected behaviour.

10.3.2 Packet Stream

The `packet_stream` is validated to check if it exists. This ensures that a valid packet stream has been chosen for input. This validation is required because CHERI Networking will hang indefinitely if an invalid packet stream is passed into the application. Therefore, the run script should detect if a packet stream is invalid and raise an error stating that the packet stream was not found. Figure 10.3 below shows the response generated when a non-existent packet stream is used in the run script.



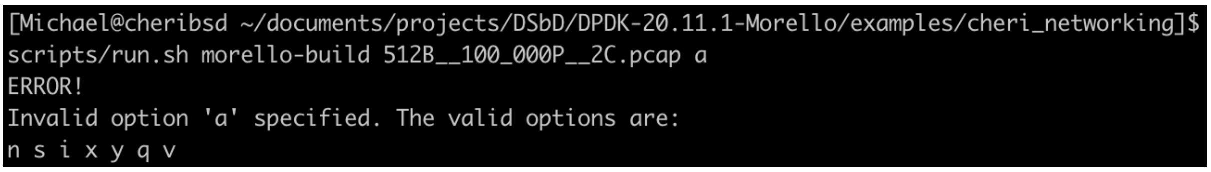
```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/examples/cheri_networking]$
scripts/run.sh morello-build invalid.pcap s
ERROR!
Packet stream '/usr/home/Michael/documents/projects/DSbD/DPDK-20.11.1-Morello/examples/cheri_
networking/packet_streams/invalid.pcap' not found!
```

Figure 10.3: A missing packet stream caught by the run script.

Figure 10.3 shows that the missing packet stream is successfully detected by the run script and an error explaining that the packet stream is not found is raised. This matches the expected behaviour.

10.3.3 Application Options

The `options` are validated to ensure that they are all implemented in CHERI Networking. The allowed options relate to the CLI arguments shown in Section 6.3, except without the corresponding hyphens. Therefore, the run script should detect if an invalid option is specified and raise an error stating which option is invalid. Figure 10.4 below shows the response generated when an invalid option is used in the run script.



```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/examples/cheri_networking]$
scripts/run.sh morello-build 512B__100_000P__2C.pcap a
ERROR!
Invalid option 'a' specified. The valid options are:
n s i x y q v
```

Figure 10.4: An invalid CHERI Networking option caught by the run script.

Figure 10.4 shows that the invalid application option is successfully detected by the run script and an error stating which option is invalid is raised. This matches the expected behaviour.

10.4 Progress Reflection

This work was unplanned and it took approximately a week to complete. It would have been very difficult to predict the need for this work in advance because it was conceptualised due to shortcomings in the DPDK framework and CHERI Networking application.

Although this work was not asked for in the project objectives, it made it easier to use CHERI Networking, adding value to the specification. It also increased the speed of development for the remainder of the work, especially regarding the recording of performance metrics which nested this script for the automatic recording of data.

11 Terminal Output

The terminal output of both CHERI Networking and the packet receiver depend on the application options specified. This section details the 3 printing options which can be used, which are default, verbose, and quiet. This was unplanned work so it does not appear on the project timeline shown in Appendix A and was not required by the project specification.

11.1 Default

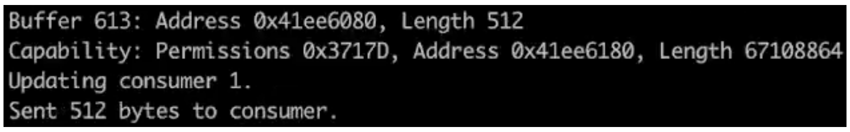
The default output is performed when neither the quiet nor verbose output modes are specified. There is no CLI argument or environment variable specifically to use the default terminal output.

11.1.1 CHERI Networking

The default terminal output of CHERI Networking will display data relating to each packet that is read into a buffer. This includes:

- The buffer that is being used to store the packet data. The buffers range from 0 to 999 because there are 1,000 buffers in the application.
- The virtual address of the packet.
- The length of the packet, in bytes.
- The permissions associated with the packet, in hexadecimal.
- The consumer that will receive the packet.
- The number of bytes sent to the consumer, only when using the IPC processing mode.

An example of the default terminal output from CHERI Networking can be seen in Figure 11.1 below.



```
Buffer 613: Address 0x41ee6080, Length 512
Capability: Permissions 0x3717D, Address 0x41ee6180, Length 67108864
Updating consumer 1.
Sent 512 bytes to consumer.
```

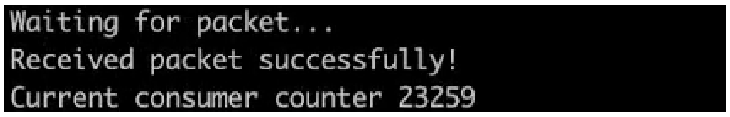
Figure 11.1: Default terminal output of CHERI Networking.

It can be observed that Figure 11.1 shows the CHERI Networking application using the IPC processing mode. This is because the number of bytes sent to the consumer is displayed. Additionally, it can be seen that the packet length is 512B, but the associated capability has a length of 67,108,864B. This is the maximum length of a CHERI capability, which is expected because capabilities should not be restricted or validated when using IPC.

11.1.2 Packet Receiver

The default terminal output of the packet receiver will display the current status of the application. This means that it will output whether the application is waiting for a packet or if it has received a packet successfully. If a packet is received successfully, the updated consumer counter is also printed to the terminal.

An example of the default terminal output from the packet receiver can be seen in Figure 11.2 below.



```
Waiting for packet...
Received packet successfully!
Current consumer counter 23259
```

Figure 11.2: Default terminal output of the packet receiver.

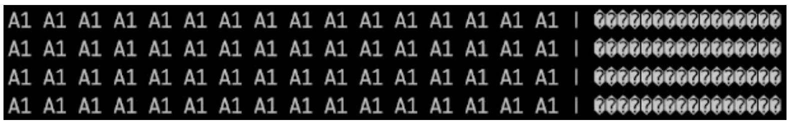
11.2 Verbose

The verbose output will display the maximum amount of useful information to the terminal. This aids the debugging process when development is active on the application. The disadvantage of verbose output is that it causes CHERI Networking to execute at a slow speed.

Verbose output is selected by either a corresponding CLI argument, `-v`, or environment variable, `PYTILIA_VERBOSE`.

The verbose output of both CHERI Networking and the packet receiver will display all of the data shown in their respective default outputs in addition to the packet data. This packet data is displayed in both hexadecimal and Unicode formats, with the hexadecimal representation of each byte on the left and the Unicode representation on the right.

An example of the verbose terminal output can be seen in Figure 11.3 below.



```
A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 | ����������������
A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 | ����������������
A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 | ����������������
A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 A1 | ����������������
```

Figure 11.3: Verbose terminal output of CHERI Networking.

The output shown in Figure 11.3 contains a small portion of the packet data stored in a 512B packet. Each byte in this example is the Unicode character corresponding to the decimal value 161, which is A1 in hexadecimal. It can be noted that the Morello terminal does not support characters outside of the ASCII character set, which ranges from 0-127. Therefore, character 161 is represented with a question mark in Figure 11.3.

11.3 Quiet

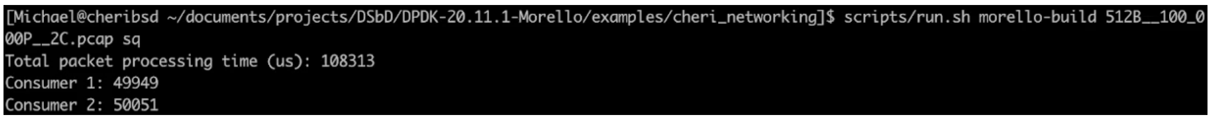
The quiet output will display the minimum required amount of information to the terminal. This is useful because it allows the application to operate at maximum speed. This also reduces the variation in processing time, which is primarily caused by printing characters to the terminal. Therefore, this output mode allows for more accurate performance metrics to be recorded.

Quiet output is selected by either a corresponding CLI argument, `-q`, or environment variable, `PYTILIA_QUIET`.

11.3.1 CHERI Networking

The quiet output of CHERI Networking will display no packet data, metadata, or debug information. The only information displayed will be the total packet processing time in microseconds and the final consumer counters.

An example of the quiet terminal output for CHERI Networking can be seen in Figure 11.4 below.



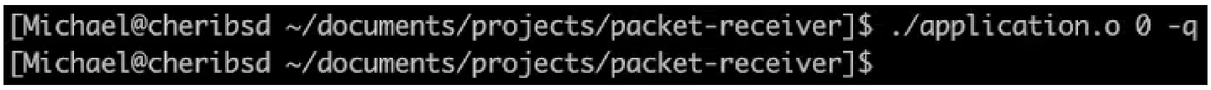
```
[Michael@cheribsd ~/documents/projects/DSbD/DPDK-20.11.1-Morello/examples/cheri_networking]$ scripts/run.sh morello-build 512B__100_0
00P__2C.pcap sq
Total packet processing time (us): 108313
Consumer 1: 49949
Consumer 2: 50051
```

Figure 11.4: Quiet terminal output of CHERI Networking.

11.3.2 Packet Receiver

The quiet output of the packet receiver will display no information at all. This is because the consumer counters are sent to the CHERI Networking application to be printed, so there is no need for duplication of this information when minimising the terminal output.

An example of the quiet terminal output for the packet receiver can be seen in Figure 11.5 below.



```
[Michael@cheribsd ~/documents/projects/packet-receiver]$ ./application.o 0 -q
[Michael@cheribsd ~/documents/projects/packet-receiver]$
```

Figure 11.5: Quiet terminal output of the packet receiver.

11.4 Progress Reflection

This work was unplanned and it took approximately a week to complete. It was not considered in advance that printing to the terminal would slow down the processing completed by the application, but as development progressed this was determined to be a significant factor.

This work was not asked for in the project objectives. However, it increased the flexibility of the applications because the quiet mode enabled performance metrics to be tested easily. Furthermore, the verbose mode made it easier to debug the application during development.

12 Performance Evaluation

This section covers the analysis of performance metrics for each operation mode of the packet processing application. This relates to task 7 of the project timeline shown in Appendix A and fulfils project objective 7 from the specification.

12.1 Overview

A series of Bash and Python scripts were created to automatically record the performance metrics for each mode of operation of CHERI Networking. The average packet processing latency, total CPU utilisation, and execution time of the application were measured and written to a CSV file. The application was run in quiet mode to remove any noise in the results which would be generated by printing to the terminal.

The scripts used the packet streams detailed in Section 9.3 as input for all 3 operation modes of CHERI Networking. This means that the packet count varied between 20,000 and 200,000 in increments of 20,000. Packet size was intended to be varied between 128B and 8KiB in powers of 2, with a constant packet count of 100,000. However, the streams containing packets of size 128B and 256B could not be used when recording the performance metrics, as discussed in Section 9.3.3. Therefore, the packet size was only varied between 512B and 8KiB.

Every input was run 10 times for each operation mode. This allowed for a reliable mean and standard deviation to be calculated, which was performed in a separate Python script. This script also plotted corresponding graphs with these statistics.

Every iteration of testing was followed by a one-second delay. This ensured there was sufficient time to complete any background processes that the CPU may have been running after the execution finished. Therefore, the execution of one iteration could not affect the performance metrics which were recorded for the next iteration.

The Bash and Python scripts used for recording the performance metrics are contained in the git repository of CHERI Networking inside the `scripts` directory, which is available on both GitLab [35] and GitHub [36]. Similarly, the Python script used for plotting graphs as well as the CSV files containing the results are available in a separate git repository on both GitLab [50] and GitHub [51].

12.2 Packet Processing Latency

In this section, the average packet processing latency is analysed according to variations in packet size and packet count.

12.2.1 Packet Size

The variation in packet processing latency with packet size is shown in Figures 12.1 and 12.2 below.

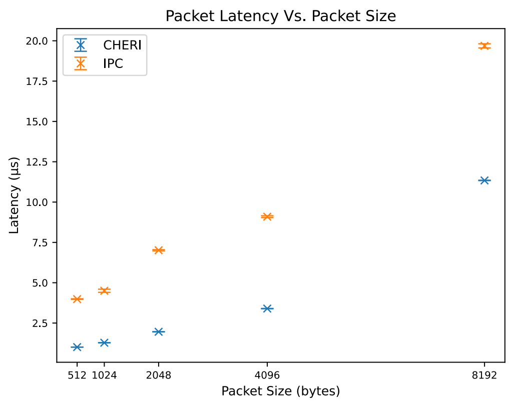


Figure 12.1: Packet processing latency against packet size for CHERI and IPC modes of operation.

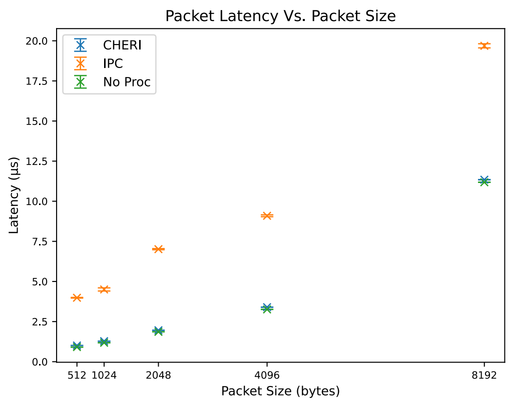


Figure 12.2: Packet processing latency against packet size for all 3 modes of operation.

Figure 12.1 shows that CHERI produces lower packet processing latency than IPC for all packet sizes. As packet size increases, the latency also increases for both modes of operation. This is the expected outcome for IPC because packets are transferred one data bit at a time. Therefore, larger packets should require more time to be sent to a consumer. However, this is not the expected outcome with CHERI capabilities. This is because the CHERI capability associated with the packet data is always going to be 129 bits long. Therefore, the latency when transferring CHERI capabilities should remain at a constant value because the same amount of data is transferred to the consumer regardless of packet size.

Figure 12.2 shows that the no-processing mode of operation produces a packet processing latency which is similar to the CHERI mode of operation. In theory, there should be no latency whenever no packets are processed. However, the time taken to read a packet into a corresponding buffer is inside the same section of code as dispatching the packet to the intended consumer. This means that the total latency recorded by the program includes the time taken to write the packet to the buffer. Therefore, a non-zero latency is calculated for the no-processing mode of operation because the program will write the packet data to a buffer before clearing the buffer. This functionality is constant for all 3 modes of operation. Therefore, the average latency calculated when not processing packets can be subtracted from the corresponding average latency calculated for both CHERI and IPC modes of operation. This results in corrected values of latency which explicitly measure the packet processing latency without external noise in the results.

Figure 12.3 below shows the corrected packet processing latencies for the CHERI and IPC modes of operation.

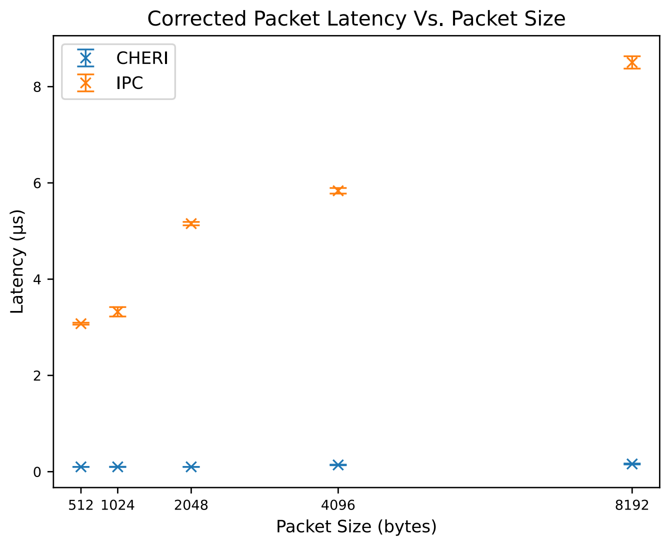


Figure 12.3: Corrected packet processing latency against packet size.

Figure 12.3 shows that as packet size increases, latency for the IPC mode of operation increases as expected. However, the CHERI-based mode of operation shows a constant latency. This differs from the uncorrected results shown previously and matches the expectation of a constant latency. Therefore, it can be observed that the benefit of using CHERI capabilities over a traditional IPC methodology is greater for larger packet sizes.

12.2.2 Packet Count

The variation in packet processing latency with packet count is shown in Figures 12.4 and 12.5 below.

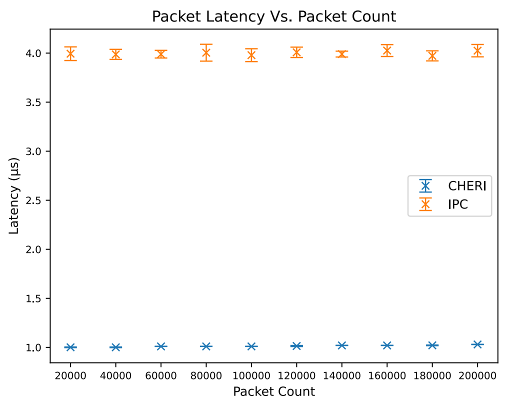


Figure 12.4: Packet processing latency against packet count for CHERI and IPC modes of operation.

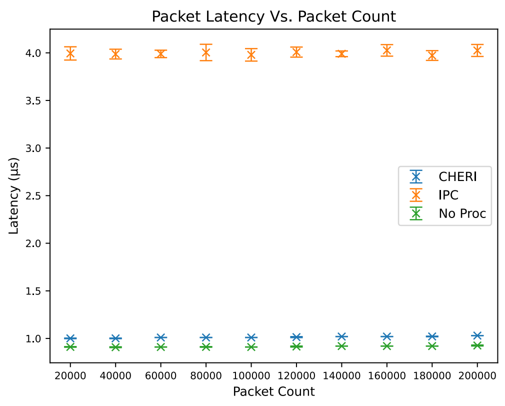


Figure 12.5: Packet processing latency against packet count for all 3 modes of operation.

Figure 12.4 shows that CHERI produces lower packet processing latency than IPC for all packet counts. This is because CHERI capabilities are 129 bits in length whereas the packets are 512B in length.

Figure 12.5 shows that latency remains constant for all 3 modes of operation irrespective of packet count. This is because all packets in these streams are the same length of 512B. Therefore, the same amount of data is written into a buffer and then transmitted for each packet, which results in a constant average latency.

The corrected packet processing latencies were calculated through the same method as in Section 12.3.1. Figure 12.6 below shows these corrected latencies for the CHERI and IPC modes of operation.

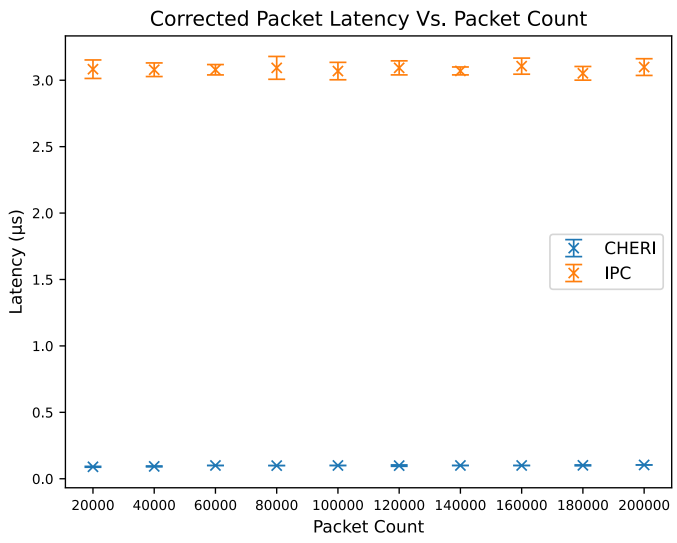


Figure 12.6: Corrected packet processing latency against packet count.

Figure 12.6 shows that as packet count increases, the latency in both modes of operation remains constant. This is the same trend as the uncorrected latencies shown in Figure 12.4. However, it can be noted that the magnitude of the corrected latencies are significantly lower than the uncorrected latencies. This is because the corrected latencies remove the delay associated with reading packets into the buffers.

12.3 CPU Utilisation

In this section, the total CPU utilisation of the packet processing application is analysed according to variations in packet size and packet count.

12.3.1 Packet Size

The variation in total CPU utilisation with packet size is shown in Figures 12.7 and 12.8 below.

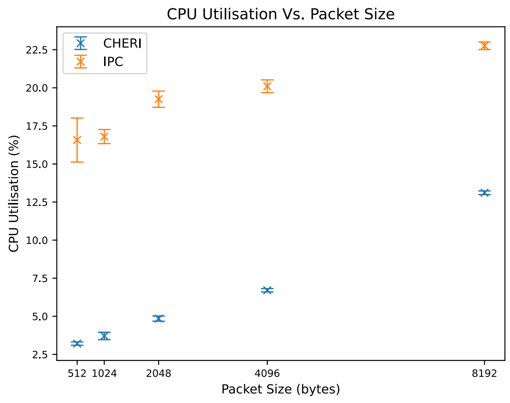


Figure 12.7: Total CPU utilisation against packet count for CHERI and IPC modes of operation.

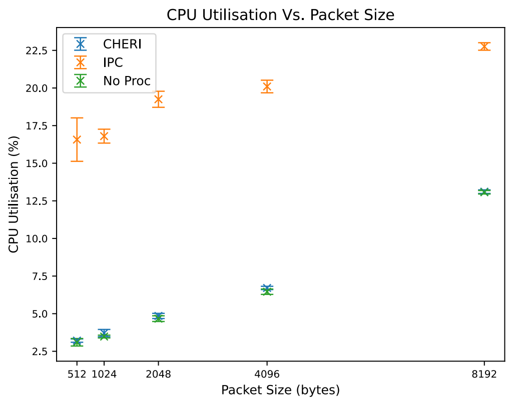


Figure 12.8: Total CPU utilisation against packet count for all 3 modes of operation.

As can be seen from Figure 12.7, the CPU utilisation increases with packet size for both the CHERI and IPC modes of operation. However, CPU utilisation increases faster for IPC than CHERI. This is because more data is transferred when using IPC. Figure 12.8 shows that the no-processing mode of operation also utilises a higher percentage of the CPU as packet size increases. This is because reading larger packets into a buffer uses more of the CPU.

12.3.2 Packet Count

The variation in total CPU utilisation with packet count is shown in Figures 12.9 and 12.10 below.

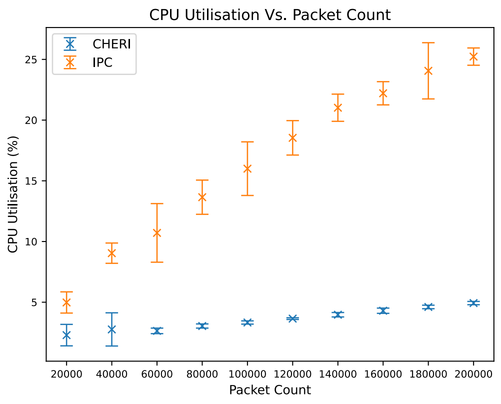


Figure 12.9: Total CPU utilisation against packet count for CHERI and IPC modes of operation.

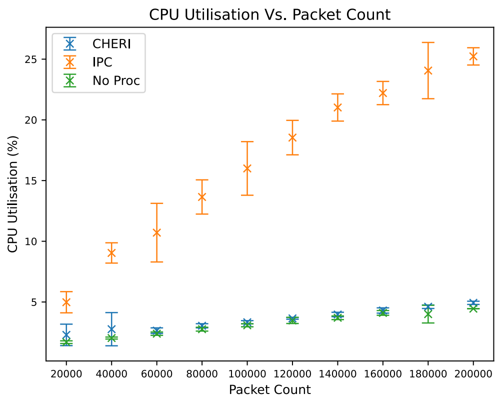


Figure 12.10: Total CPU utilisation against packet count for all 3 modes of operation.

As can be seen from Figure 12.9, the CPU utilisation increases with packet count for both the CHERI and IPC modes of operation. It can also be noted that CPU utilisation increases faster for IPC than CHERI. Additionally, the standard deviation is smaller for CHERI compared to IPC. This is because passing a CHERI capability is done in one clock cycle on the CPU, whereas every byte of packet data must be sent separately when using UDP sockets. This provides more opportunity for noise to affect the results. Figure 12.10 shows that the no-processing mode of operation also utilises a higher percentage of the CPU as packet count increases. This is because more CPU is required to read more packets into a buffer.

12.4 Execution Time

In this section, the total execution time of the packet processing application is analysed as packet size and packet count are varied. This was not required in the project specification but it was determined to be beneficial because it demonstrates how statistically significant the total packet processing time is relative to the time taken for the whole application to complete execution.

12.4.1 Packet Size

The variation in execution time with packet size is shown in Figure 12.11 below.

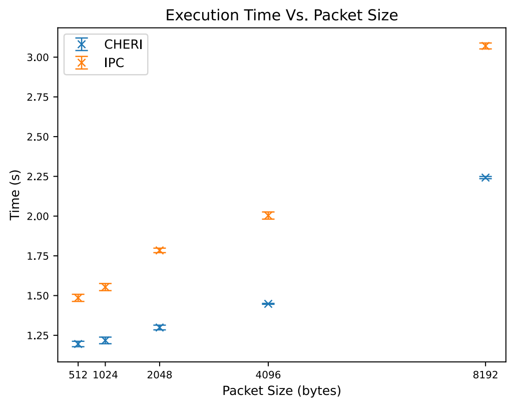


Figure 12.11: Execution time against packet size for the CHERI and IPC modes of operation.

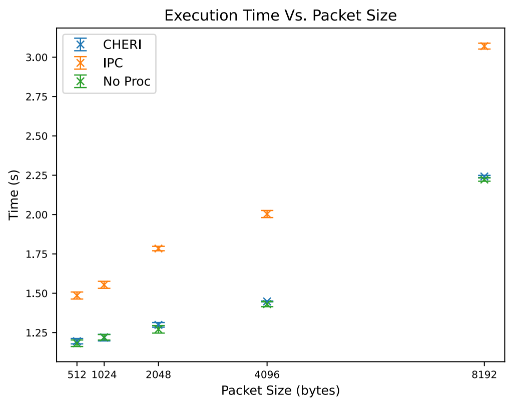


Figure 12.12: Execution time against packet size for all 3 modes of operation.

As can be seen in Figure 12.11, using CHERI capabilities is faster than using IPC for all packet sizes. The execution time of the application increases as packet size increases for both the CHERI and IPC modes of operation. However, the execution time increases at a greater rate when using IPC compared to CHERI. This is because the average packet processing latency increases for IPC whereas it is constant for CHERI capabilities. If the packet processing latency increases, then the execution time should also increase. Figure 12.12 shows that the execution time also increases with packet size for the no-processing mode of operation. This is because the application takes longer to read larger packets into buffers, which affects all 3 modes of operation.

When comparing the execution time for all 3 modes of operation, it can be seen that using CHERI capabilities requires approximately the same amount of time as performing no processing on the packets. This means that the packet processing latency associated with using CHERI capabilities is statistically insignificant relative to the total packet processing time. However, IPC requires more time to complete execution and it can be observed that packet processing contributes a statistically significant amount of time to the total execution time of the application.

12.4.2 Packet Count

The variation in execution time with packet count is shown in Figure 12.13 below.

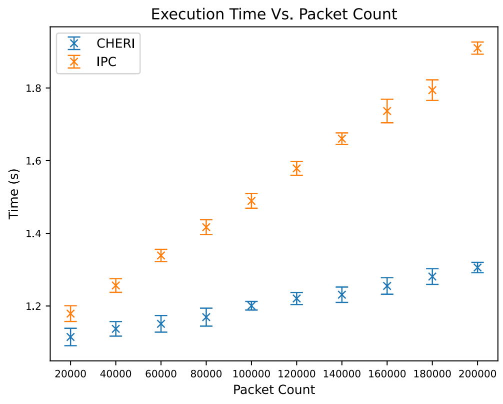


Figure 12.13: Execution time against packet count for the CHERI and IPC modes of operation.

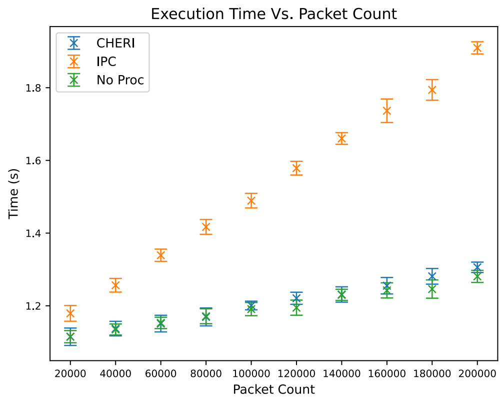


Figure 12.14: Execution time against packet count for all 3 modes of operation.

As can be seen in Figure 12.13, using CHERI capabilities is faster than using IPC for all packet counts. The execution time of the application also increases with packet count for both the CHERI and IPC modes of operation. This is because every packet is processed sequentially, not in parallel. This means that every packet processed will increase the total execution time. However, the execution time increases at a greater rate when using IPC compared to CHERI. This is because the average packet processing latency is higher for IPC compared to CHERI. Therefore, the execution time increases at a greater rate according to packet count when using IPC. Figure 12.14 shows that the execution time increases for the no-processing mode of operation at a rate which is comparable to CHERI. This proves that the packet processing time remains statistically insignificant when using CHERI capabilities and also remains statistically significant when using IPC to process packets.

12.5 Progress Reflection

Task 7 of the project timeline shown in Appendix A refers to recording and analysing the performance metrics of the packet processing application. Two weeks were allocated to this task, with one week to record the results and another week to plot graphs and analyse the results. Recording the results took one week as expected. However, when analysing the results initially, the packet processing latencies appeared to be very large relative to what was expected. This resulted in the discovery of a large amount of noise in the results due to the packets printing to the terminal. This led to the work relating to the quiet and verbose options of CHERI Networking and the packet receiver detailed in Section 6.3. The results were then rerecorded the following week with the quiet option enabled. Furthermore, when analysing the new results, it was noticed that the packet processing latency was not constant for the CHERI mode of operation as expected. This was the inspiration for the idea of developing a no-processing mode of operation for CHERI Networking to provide a series of control readings and calculate corrected metrics. All of this work was unplanned in the timeline and, in total, required an extra two weeks of work. This was possible because this was the last developmental task for this project and all other work was completed by this stage, except for the presentations and reports for QUB and the industrial partners. Therefore, the contingency time that was initially allocated at the end of the project schedule was used to complete these tasks. In hindsight, this extra work could not have been planned for in advance because it was discovered through the initial results recorded towards the end of project development. Therefore, this was an ideal use of the contingency time.

Objective 7 of the project specification only asks for the packet processing latency and total CPU utilisation to be evaluated for CHERI and IPC processing modes. Therefore, the inclusion of a no-processing mode to provide a series of control measurements adds value to the specification. Furthermore, the analysis of total execution time relative to the total packet processing time provides extra insights than what was required.

13 Discussion and Conclusions

This section discusses the work outlined in this report, including the conclusions reached from the completed work. A reflection on the project objectives and expected timeline are given as well as a discussion on the potential future work relating to this project.

13.1 Conclusions

In this report, the creation of a packet processing application called CHERI Networking was detailed. This application implements digital security by design (DSbD) principles, including capability hardware-enhanced RISC instructions (CHERI). This application dispatches packets to consumers through either CHERI capabilities or User Datagram Protocol (UDP) sockets. It was shown that CHERI capabilities provide a secure method of packet processing when interacting with untrusted, third-party consumer plugins. The capability bounds and permissions were both restricted to remove any potential security vulnerabilities caused by malicious or poorly written code. The capability validation was also enforced in a way which improves over the previous implementation by Pytilia because there is no maximum number of packets which can be reliably processed.

Additionally, DSbD principles were shown to provide performance improvements compared to a traditional inter-process communications (IPC) methodology when dispatching packets to consumer plugins, regarding packet processing latency, total CPU utilisation, and total program execution time. It was shown that CHERI capabilities provide consistent latency for different packet sizes and counts, whereas IPC produces latency which does not vary with packet count, but increases with packet size. It was also shown that latency is responsible for a very small proportion of the total execution time of a packet processing application when using CHERI capabilities, whereas it is a significant proportion when using IPC. This proved that the performance benefits of using CHERI are increased when using larger packets.

Additionally, a successful port of the Data Plane Development Kit (DPDK) framework for the Morello board was discussed. This port is sufficient for creating a functional networking application on the Morello board and permits the use of CHERI capabilities. However, an application created with this port will be limited to using a single process.

13.2 Project Objectives

All of the project objectives from the specification were achieved in addition to some extra work. A run script was created to simplify the usage of the CHERI Networking and provide extra validation checks. Additionally, more performance metrics were recorded for CHERI Networking than required. Metrics were recorded for packet streams varying in both packet count and size. The total execution time was recorded in addition to the required packet processing latency and total CPU utilisation. A no-processing mode was added to CHERI Networking to provide a control measure when comparing the performance metrics. A quiet mode was also added to remove the noise in the recorded metrics caused by printing to the terminal.

13.3 Progress Reflection

All the tasks planned for this project according to the plan given in Appendix A were completed in addition to some unplanned tasks. However, the time allocated to each task was not always accurate. For example, the DPDK port was planned for 3 weeks but it took approximately a month to complete. On the other hand, the packet processing application was allocated 8 weeks but was completed in approximately 5 weeks. However, these inaccuracies mostly averaged out to allow the developmental work to complete at a similar date to what was expected. The schedule also provided some contingency time, which was mainly used for completing the additional tasks that were not planned for. In hindsight, these additional tasks provided added value to the project but it may have been more beneficial to spend this contingency time focusing exclusively on the presentations and reports required for both university and the industrial partners. Overall, the plan contained an underlying issue, which was that it did not account for assessments which were required for other modules. This caused the number of hours dedicated to this project to vary every week. This allowed for more time to be invested in this project at the beginning of January, allowing for increased productivity. However, this caused delays at the end of February and March, which negatively impacted the closing stages of this project.

13.4 Industrial Partners

This project earned a presentation slot at the DSbD demonstration day in London on Wednesday 22nd March 2023. Digital Catapult states in the DSbD Demo Day booklet "When comparing like-for-like applications implemented using traditional and DSbD approaches, the Pytilia team has seen that the DSbD approach delivers improved performance (for both latency and CPU utilisation) and the security benefits associated with CHERI capabilities" [52].

Pytilia also viewed the project as a success. When asked for comment, Pytilia stated "In successfully completing this project, Michael demonstrated extensive knowledge of hardware, system software, cybersecurity and networking domains and interacted professionally with the Pytilia R&D team, the DSbD programme team and representatives of several technology partners. Michael's work showed that the CHERI technologies underlying DSbD are broadly applicable to a range of core components in the critical IT infrastructure space and can eliminate security-performance tradeoffs which are widespread today."

Overall, this shows that the project was viewed as a success by both industrial partners.

13.5 Future Work

This project proved that DSbD initiatives can be used securely in networking applications which communicate with untrusted, third-party plugins. However, CHERI Networking and the previous application developed by Pytilia, Limelight, are the only two networking applications created to facilitate the use of DSbD initiatives. Therefore, more advanced networking applications could be created which use CHERI capabilities to interact with such plugins to provide performance improvements. Examples of applications which could benefit from this include firewalls, network monitors, and storage backup.

Additionally, the DPDK port created in this project is limited to using a single process due to kernel modules which are missing from CheriBSD. The required kernel modules, `contigmem` and `nic_uio`, could be ported to CheriBSD. Corresponding changes could be made to the DPDK port which would remove this limitation and allow for multiple processes to be used.

14 References

[1] D. S. by Design, “Digital security by design,” Sep 2022, accessed: 2023-03-31. [Online]. Available: https://www.dsbd.tech/

[2] Wikipedia, “Packet processing,” Apr 2022, accessed: 2023-04-01. [Online]. Available: https://en.wikipedia.org/wiki/Packet_processing/

[3] P. A. Network, “Packet flow sequence in pan-os,” Jun 2021, accessed: 2023-04-01. [Online]. Available: https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000ClVHCA0

[4] Microsoft, “Packet monitor (pktmon),” May 2022, accessed: 2023-04-01. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/networking/technologies/pktmon/pktmon

[5] S. et al., “Computer communications,” Sep 2018, accessed: 2023-04-01. [Online]. Available: https://doi.org/10.1016/j.comcom.2018.07.014

[6] W. Beginner, “6 best wordpress firewall plugins compared,” Nov 2020, accessed: 2023-04-01. [Online]. Available: https://www.wpbeginner.com/plugins/best-wordpress-firewall-plugins-compared/

[7] BlogVault, “9 best wordpress firewall plugins (compared),” Jan 2022, accessed: 2023-04-01. [Online]. Available: https://blogvault.net/best-wordpress-firewall-plugins/

[8] D. S. by Design, “Pytilia - digital security by design,” Feb 2022, accessed: 2023-03-30. [Online]. Available: https://www.dsbd.tech/success-story/pytilia/

[9] R. Sheldon, “What is a virtual address and how does it work?” Feb 2023, accessed: 2023-04-01. [Online]. Available: https://www.techtarget.com/whatis/definition/virtual-address

[10] Wikipedia, “Digital catapult,” Mar 2023, accessed: 2023-03-30. [Online]. Available: https://en.wikipedia.org/wiki/Digital_Catapult

[11] P. Limited, “Pytilia - software development,” Sep 2022, accessed: 2023-03-30. [Online]. Available: https://pytilia.io/

[12] E. O'Reilly, "pytilia/dpdk-v20.11.1," Oct 2021, accessed: 2023-04-11. [Online]. Available: https://github.com/pytilia/DPDK-v20.11.1

[13] M. Allen, "Applicability of digital security by design to performance-sensitive networking applications," Jan 2023, eLE3001 Interim Report.

[14] R. N. M. Watson, "Department of computer science and technology: Capability hardware enhanced risc instructions (cheri)," Jan 2022, accessed: 2023-03-31. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[15] D. et al., "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment," University of Cambridge Computer Laboratory, Tech. Rep., Apr 2019, accessed: 2023-03-31. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201904-asplos-cheriabi.pdf

[16] D. S. by Design, "How it works - digital security by design," Sep 2022, accessed: 2023-03-31. [Online]. Available: https://www.dsbd.tech/how-it-works/

[17] H. Almatary, "Cheri compartmentalization for embedded systems," University of Cambridge Computer Laboratory, Tech. Rep. 976, Nov 2022. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-976.pdf

[18] Arm, "Morello program - arm," Jan 2023, accessed: 2023-03-31. [Online]. Available: https://www.arm.com/architecture/cpu/morello

[19] ——, "Morello fixed virtual platform models," 2022, accessed: 2023-03-31. [Online]. Available: https://developer.arm.com/documentation/102233/2022-0M0/Introduction-to-Arm-Development-Studio-Morello-Edition/Morello-Fixed-Virtual-Platform-models

[20] R. N. M. Watson, "Cheribsd," 2022, accessed: 2023-03-31. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html

[21] U. of Cambridge, "Github - ctrsd-cheri/llvm-project," Jan 2023, accessed: 2023-03-31. [Online]. Available: https://github.com/CTSRD-CHERI/llvm-project

[22] W. et al., “Cheri c/c++ programming guide,” University of Cambridge, Tech. Rep. 947, Jun 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

[23] U. of Cambridge, “Github - ctrsd-cheri/gdb,” Nov 2022, accessed: 2023-03-31. [Online]. Available: https://github.com/CTSRD-CHERI/gdb

[24] D. Project, “About - dpdk,” Nov 2022, accessed: 2023-04-01. [Online]. Available: https://www.dpdk.org/about/

[25] K. Obiidykhata, “What is dpdk, and what is it used for?” Feb 2021, accessed: 2023-04-01. [Online]. Available: https://blog.cloudlinux.com/what-is-dpdk-and-what-is-it-used-for/

[26] D. Project, “Supported hardware,” Mar 2023, accessed: 2023-04-01. [Online]. Available: https://core.dpdk.org/supported/

[27] ——, “Getting started guide for windows,” Mar 2023, accessed: 2023-04-01. [Online]. Available: http://doc.dpdk.org/guides/windows_gsg/index.html

[28] ——, “Getting started guide for freebsd,” Mar 2023, accessed: 2023-04-01. [Online]. Available: http://doc.dpdk.org/guides/freebsd_gsg/index.html

[29] ——, “Getting started guide for linux,” Mar 2023, accessed: 2023-04-01. [Online]. Available: http://doc.dpdk.org/guides/linux_gsg/index.html

[30] A. Limited, “Morello development platform and software getting started guide,” Aug 2022, accessed: 2023-04-01. [Online]. Available: https://developer.arm.com/documentation/den0132/0100/Flash-the-onboard-SD-card

[31] M. Allen, “Student grades - gitlab,” Nov 2022, accessed: 2023-04-10. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40266651/student-grades

[32] ——, “Michaelta0111/student-grades,” Nov 2022, accessed: 2023-04-10. [Online]. Available: https://github.com/MichaelTA0111/Student-Grades

[33] DPDK, “5. eal parameters,” Mar 2023, accessed: 2023-04-12. [Online]. Available: https://doc.dpdk.org/guides/freebsd_gsg/freebsd_eal_parameters.html

[34] ——, “3. compiling the dpdk target from source,” Mar 2023, accessed: 2023-04-12. [Online]. Available: http://doc.dpdk.org/guides/freebsd_gsg/build_dpdk.html

[35] M. Allen, “cheri-networking - gitlab,” Apr 2023, accessed: 2023-04-05. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40266651/cheri_networking

[36] ——, “Michaelta0111/cheri-networking,” Apr 2023, accessed: 2023-04-05. [Online]. Available: https://github.com/MichaelTA0111/CHERI-Networking

[37] Pico, “Udp vs tcp,” 2023, accessed: 2023-04-16. [Online]. Available: https://www.pico.net/kb/udp-vs-tcp/

[38] locall.host, “Introduction to localhost 5000 and its purpose,” Mar 2023, accessed: 2023-04-14. [Online]. Available: https://locall.host/5000/

[39] T. Point, “Ipv6 - headers,” Apr 2023, accessed: 2023-04-07. [Online]. Available: https://www.tutorialspoint.com/ipv6/ipv6_headers.htm

[40] M. Allen, “packet-receiver - gitlab,” Apr 2023, accessed: 2023-04-12. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40266651/packet-receiver

[41] ——, “Michaelta0111/packet-receiver,” Feb 2023, accessed: 2023-04-12. [Online]. Available: https://github.com/MichaelTA0111/Packet-Receiver

[42] ——, “Packet generator - gitlab,” Apr 2023, accessed: 2023-04-03. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40266651/packet-generator

[43] ——, “Michaelta0111/packet-generator,” Apr 2023, accessed: 2023-04-03. [Online]. Available: https://github.com/MichaelTA0111/Packet-Generator

[44] GitHub, “About large files on github,” Apr 2023, accessed: 2023-04-12. [Online]. Available: https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github

[45] D. Newman, “Picking the ’best’ packet size,” Nov 2006, accessed: 2023-04-03. [Online]. Available: https://www.networkworld.com/article/2300175/picking-the--best--packet-size.html

[46] E. P., “What is a udp socket?” Mar 2023, accessed: 2023-04-14. [Online]. Available: https://www.easytechjunkie.com/what-is-a-udp-socket.htm

[47] Q. T. Luu, “Retransmission rules for tcp,” Jan 2023, accessed: 2023-04-14. [Online]. Available: https://www.baeldung.com/cs/tcp-retransmission-rules

[48] M. Allen, “Cheri-networking/run.sh,” Feb 2023, accessed: 2023-04-13. [Online]. Available: https://github.com/MichaelTA0111/CHERI-Networking/blob/master/scripts/run.sh

[49] ——, “scripts/run.sh,” Feb 2023, accessed: 2023-04-13. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40266651/cheri_networking/-/blob/main/scripts/run.sh

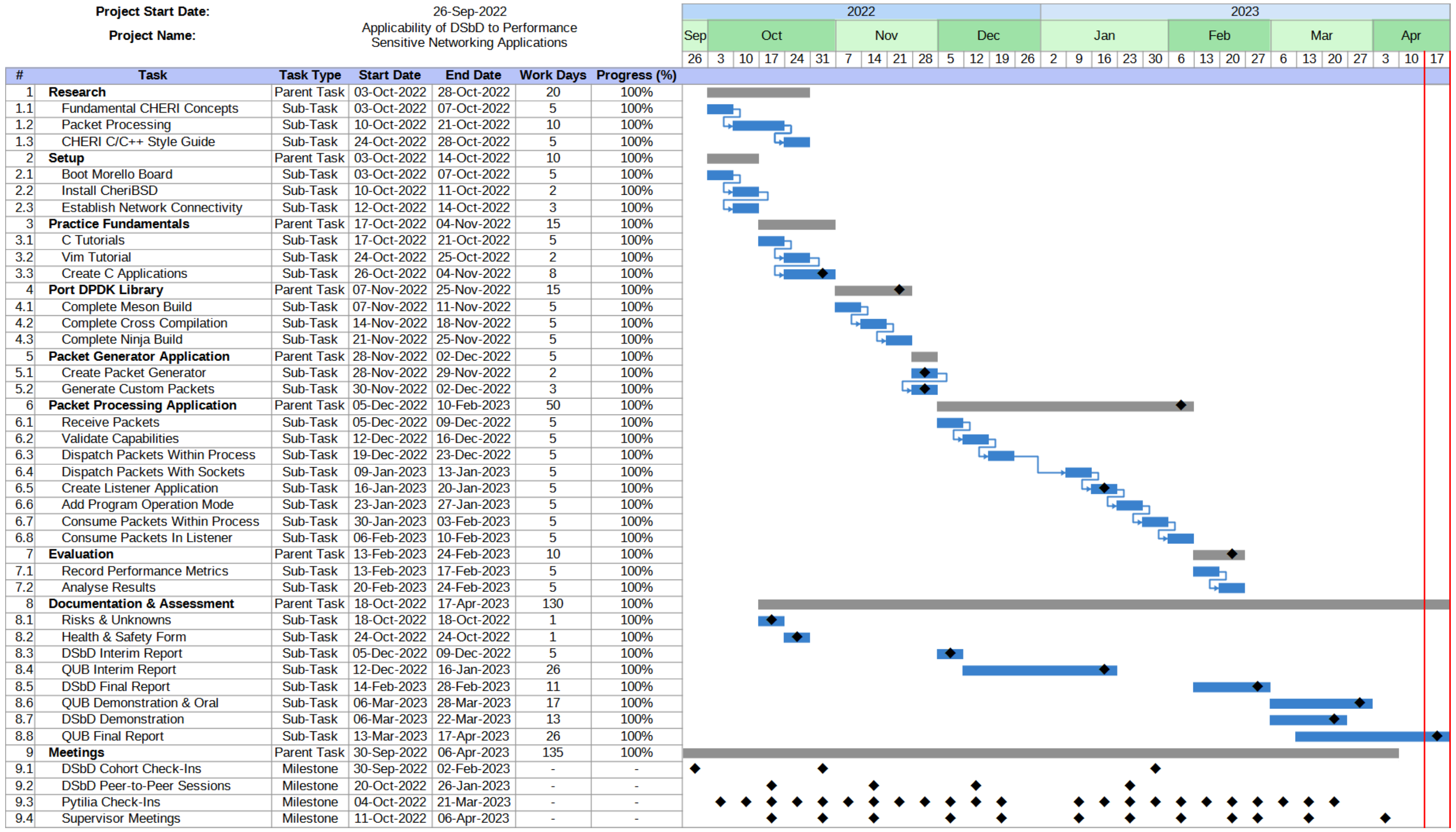
[50] ——, “Result plotter - gitlab,” Apr 2023, accessed: 2023-04-05. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/40266651/result-plotter

[51] ——, “Michaelta0111/result-plotter,” Apr 2023, accessed: 2023-04-05. [Online]. Available: https://github.com/MichaelTA0111/Result-Plotter

[52] D. Catapult, “Demo day,” March 2023, accessed: 2023-04-17. [Online]. Available: https://dsbd.tech/wp-content/uploads/2023/03/DSBD-TAP-Demo_Day_booklet_Online_version_230310.pdf

Appendix A Project Plan

The Gantt chart for the project, updated with progress as of 17th April 2023, is shown in Figure A.1 below.



Project Start Date: 26-Sep-2022

Project Name: Applicability of DSbD to Performance Sensitive Networking Applications

#	Task	Task Type	Start Date	End Date	Work Days	Progress (%)
1	**Research**	Parent Task	03-Oct-2022	28-Oct-2022	20	100%
1.1	Fundamental CHERI Concepts	Sub-Task	03-Oct-2022	07-Oct-2022	5	100%
1.2	Packet Processing	Sub-Task	10-Oct-2022	21-Oct-2022	10	100%
1.3	CHERI C/C++ Style Guide	Sub-Task	24-Oct-2022	28-Oct-2022	5	100%
2	**Setup**	Parent Task	03-Oct-2022	14-Oct-2022	10	100%
2.1	Boot Morello Board	Sub-Task	03-Oct-2022	07-Oct-2022	5	100%
2.2	Install CheriBSD	Sub-Task	10-Oct-2022	11-Oct-2022	2	100%
2.3	Establish Network Connectivity	Sub-Task	12-Oct-2022	14-Oct-2022	3	100%
3	**Practice Fundamentals**	Parent Task	17-Oct-2022	04-Nov-2022	15	100%
3.1	C Tutorials	Sub-Task	17-Oct-2022	21-Oct-2022	5	100%
3.2	Vim Tutorial	Sub-Task	24-Oct-2022	25-Oct-2022	2	100%
3.3	Create C Applications	Sub-Task	26-Oct-2022	04-Nov-2022	8	100%
4	**Port DPDK Library**	Parent Task	07-Nov-2022	25-Nov-2022	15	100%
4.1	Complete Meson Build	Sub-Task	07-Nov-2022	11-Nov-2022	5	100%
4.2	Complete Cross Compilation	Sub-Task	14-Nov-2022	18-Nov-2022	5	100%
4.3	Complete Ninja Build	Sub-Task	21-Nov-2022	25-Nov-2022	5	100%
5	**Packet Generator Application**	Parent Task	28-Nov-2022	02-Dec-2022	5	100%
5.1	Create Packet Generator	Sub-Task	28-Nov-2022	29-Nov-2022	2	100%
5.2	Generate Custom Packets	Sub-Task	30-Nov-2022	02-Dec-2022	3	100%
6	**Packet Processing Application**	Parent Task	05-Dec-2022	10-Feb-2023	50	100%
6.1	Receive Packets	Sub-Task	05-Dec-2022	09-Dec-2022	5	100%
6.2	Validate Capabilities	Sub-Task	12-Dec-2022	16-Dec-2022	5	100%
6.3	Dispatch Packets Within Process	Sub-Task	19-Dec-2022	23-Dec-2022	5	100%
6.4	Dispatch Packets With Sockets	Sub-Task	09-Jan-2023	13-Jan-2023	5	100%
6.5	Create Listener Application	Sub-Task	16-Jan-2023	20-Jan-2023	5	100%
6.6	Add Program Operation Mode	Sub-Task	23-Jan-2023	27-Jan-2023	5	100%
6.7	Consume Packets Within Process	Sub-Task	30-Jan-2023	03-Feb-2023	5	100%
6.8	Consume Packets In Listener	Sub-Task	06-Feb-2023	10-Feb-2023	5	100%
7	**Evaluation**	Parent Task	13-Feb-2023	24-Feb-2023	10	100%
7.1	Record Performance Metrics	Sub-Task	13-Feb-2023	17-Feb-2023	5	100%
7.2	Analyse Results	Sub-Task	20-Feb-2023	24-Feb-2023	5	100%
8	**Documentation & Assessment**	Parent Task	18-Oct-2022	17-Apr-2023	130	100%
8.1	Risks & Unknowns	Sub-Task	18-Oct-2022	18-Oct-2022	1	100%
8.2	Health & Safety Form	Sub-Task	24-Oct-2022	24-Oct-2022	1	100%
8.3	DSbD Interim Report	Sub-Task	05-Dec-2022	09-Dec-2022	5	100%
8.4	QUB Interim Report	Sub-Task	12-Dec-2022	16-Jan-2023	26	100%
8.5	DSbD Final Report	Sub-Task	14-Feb-2023	28-Feb-2023	11	100%
8.6	QUB Demonstration & Oral	Sub-Task	06-Mar-2023	28-Mar-2023	17	100%
8.7	DSbD Demonstration	Sub-Task	06-Mar-2023	22-Mar-2023	13	100%
8.8	QUB Final Report	Sub-Task	13-Mar-2023	17-Apr-2023	26	100%
9	**Meetings**	Parent Task	30-Sep-2022	06-Apr-2023	135	100%
9.1	DSbD Cohort Check-Ins	Milestone	30-Sep-2022	02-Feb-2023	-	-
9.2	DSbD Peer-to-Peer Sessions	Milestone	20-Oct-2022	26-Jan-2023	-	-
9.3	Pytilia Check-Ins	Milestone	04-Oct-2022	21-Mar-2023	-	-
9.4	Supervisor Meetings	Milestone	11-Oct-2022	06-Apr-2023	-	-

Figure A.1: A Gantt chart for the complete timeline of the project.

Figure A.1 shows the Gantt chart for the project. The grey and blue bars represent completed parent and sub-tasks respectively. Milestones and meetings are marked with a diamond. The current week is marked on the Gantt chart with red vertical lines. The timeline contains approximately 4 weeks of contingency time regarding the developmental work (tasks 2-7) before the project presentations were scheduled to occur during the week commencing Monday 20th March 2023.

It can be seen from the Gantt chart that all tasks have been completed successfully. A reflection on progress against the Gantt chart schedule is detailed at the end of all relevant report sections in a corresponding subsection.

It can be noted that some tasks were scheduled to run concurrently, which is typically only possible when separate people are assigned to those tasks. However, this project was conducted by one person. This scheduling decision was made because the developmental work required the use of the Morello board, whereas the other tasks did not. The Morello board was located in Pytilia's office in Belfast and it could not be accessed remotely. This meant that no developmental work could be completed within the QUB campus or from home. To allow for work to be completed at these locations, tasks relating to research, documentation, and assessment were scheduled concurrently with developmental work. It can be noted that none of the tasks for developmental work were scheduled to run concurrently, ensuring that the schedule was not overloaded.

Appendix B CHERI Networking Options Unit Tests

Input	Expected Result	Actual Result
CLI arguments `-s -i`	IPC processing mode is selected	Works as intended
CLI arguments `-s -n`	No-processing mode is selected	Works as intended
CLI arguments `-i -n`	No-processing mode is selected	Works as intended
CLI arguments `-s -i -n`	No-processing mode is selected	Works as intended
CLI arguments `-i -x`	IPC processing mode is selected, no bounds error is raised	Works as intended
CLI arguments `-i -y`	IPC processing mode is selected, no permissions error is raised	Works as intended
CLI arguments `-n -x`	No-processing mode is selected, no bounds error is raised	Works as intended
CLI arguments `-n -y`	No-processing mode is selected, no permissions error is raised	Works as intended
CLI arguments `-s -x`	CHERI processing mode is selected, bounds error is raised	Works as intended
CLI arguments `-s -y`	CHERI processing mode is selected, permissions error is raised	Works as intended
CLI arguments `-s -x -y`	CHERI processing mode is selected, bounds error is raised	Works as intended
CLI arguments `-s -q -v`	CHERI processing mode is selected, verbose output is used	Works as intended
No CLI arguments or environment variables	Error is raised for no process type being specified	Works as intended
CLI arguments `-x -y -q -v`	Error is raised for no process type being specified	Works as intended
CLI argument `-s`, environment variable `PYTILIA_INTER_PROCESS`	CHERI processing mode is selected	Works as intended
CLI argument `-i`, environment variable `PYTILIA_NO_PROCESS`	IPC processing mode is selected	Works as intended
CLI arguments `-s -q`, environment variable `PYTILIA_VERBOSE`	CHERI processing mode is selected, quiet output is used	Works as intended
CLI arguments `-s -v`, environment variable `PYTILIA_QUIET`	CHERI processing mode is selected, verbose output is used	Works as intended

Table B.1: Unit tests for CHERI Networking application options.

Appendix C Consumer Counter Tests

C.1 CHERI Processing Mode

C.1.1 Varied Packet Size

Packet Count	Packet Length (Bytes)	Expected Counter 0	Expected Counter 1	Actual Counter 0	Actual Counter 1
100,000	128	49,998	50,002	49,998	50,002
100,000	258	50,040	49,960	50,040	49,960
100,000	512	50,146	49,854	50,146	49,854
100,000	1,024	49,957	50,043	49,957	50,043
100,000	2,048	49,781	50,219	49,781	50,219
100,000	4,096	50,013	49,987	50,013	49,987
100,000	8,192	50,200	49,800	50,200	49,800

Table C.1: Consumer counter comparison for CHERI processing mode.

C.1.2 Varied Packet Count

Packet count	Packet length (Bytes)	Expected Counter 0	Expected Counter 1		
20,000	512	10,014	9,986	10,014	9,986
40,000	512	20,042	19,958	20,042	19,958
60,000	512	30,056	29,944	30,056	29,944
80,000	512	40,095	39,905	40,095	39,905
100,000	512	50,146	49,854	50,146	49,854
120,000	512	60,223	59,777	60,223	59,777
140,000	512	70,303	69,697	70,303	69,697
160,000	512	79,671	80,329	79,671	80,329
180,000	512	89,784	90,216	89,784	90,216
200,000	512	99,749	100,251	99,749	100,251

Table C.2: Consumer counter comparison for CHERI processing mode.

C.2 IPC Processing Mode

C.2.1 Varied Packet Size

Packet Count	Packet Length (Bytes)	Expected Counter 0	Expected Counter 1	Actual Counter 0	Actual Counter 1
100,000	128	49,998	50,002	13,784	13,996
100,000	258	50,040	49,960	34,396	34,647
100,000	512	50,146	49,854	50,146	49,854
100,000	1,024	49,957	50,043	49,957	50,043
100,000	2,048	49,781	50,219	49,781	50,219
100,000	4,096	50,013	49,987	50,013	49,987
100,000	8,192	50,200	49,800	50,200	49,800

Table C.3: Consumer counter comparison for IPC processing mode.

C.2.2 Varied Packet Count

Packet count	Packet length (Bytes)	Expected Counter 0	Expected Counter 1		
20,000	512	10,014	9,986	10,014	9,986
40,000	512	20,042	19,958	20,042	19,958
60,000	512	30,056	29,944	30,056	29,944
80,000	512	40,095	39,905	40,095	39,905
100,000	512	50,146	49,854	50,146	49,854
120,000	512	60,223	59,777	60,223	59,777
140,000	512	70,303	69,697	70,303	69,697
160,000	512	79,671	80,329	79,671	80,329
180,000	512	89,784	90,216	89,784	90,216
200,000	512	99,749	100,251	99,749	100,251

Table C.4: Consumer counter comparison for IPC processing mode.