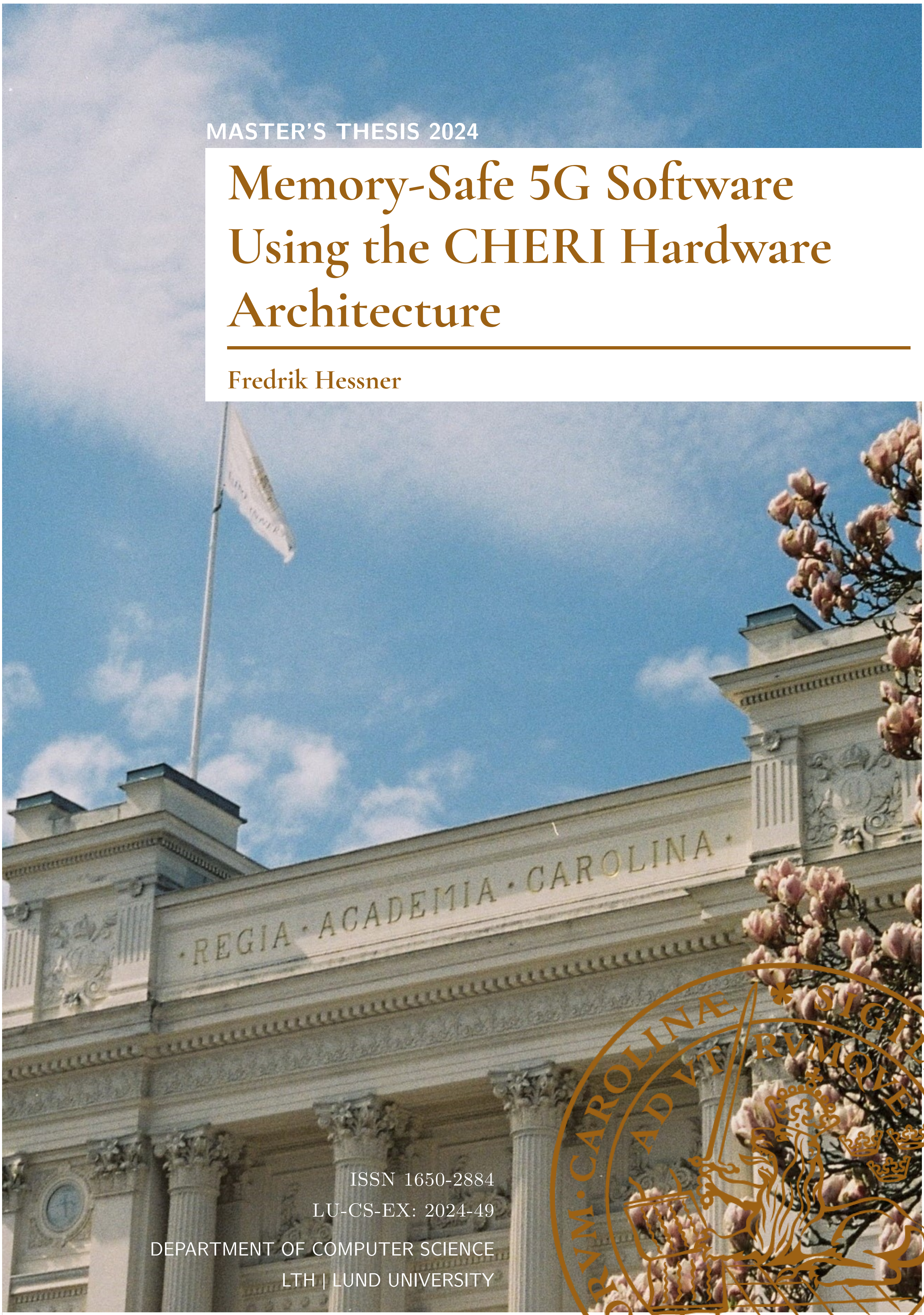


MASTER'S THESIS 2024

Memory-Safe 5G Software Using the CHERI Hardware Architecture

Fredrik Hessner

ISSN 1650-2884
LU-CS-EX: 2024-49

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-49

Memory-Safe 5G Software Using the CHERI Hardware Architecture

Minnessäker 5G-mjukvara möjliggjord av CHERI-hårdvaruarkitektur

Fredrik Hessner

Memory-Safe 5G Software Using the CHERI Hardware Architecture

(A study on how the 5G software performs on the CHERI hardware architecture)

Fredrik Hessner
`Fr2875he-s@student.lu.se`

August 8, 2024

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jonas Skeppstedt, `jonas.skeppstedt@cs.lth.se`
Håkan Englund, `hakan.englund@ericsson.se`

Examiner: Sven Robertz, `sven.robertz@cs.lth.se`

Abstract

As the problem of memory-safety continues to pose a significant threat to the software ecosystem, developers need to consider strategies to address these vulnerabilities. For high-performing applications built on the C/C++ programming languages, solutions have to be easy to adapt without compromising too much on performance. In this thesis, high-performing 5G software is evaluated using the CHERI hardware architecture designed to significantly improve memory-safety. The research was based on two metrics: The required effort to port the software and the overhead of the memory-safe architecture compared to the standard implementation. The results show that only around 1% of the total lines of code had to be re-written for the program to successfully run. The overhead of the two measurements of the benchmark were 38% and 53% compared to baseline, and are believed to come from addressable architectural limitations.

Keywords: Memory-safety,CHERI,AArch64,5G,C,C++

Acknowledgements

I would like to thank my supervisor Håkan Englund as well as Peter Svensson and Jonas Åberg at Ericsson research, without whom this thesis would not have been possible. I would also like to thank my supervisor Jonas Skeppstedt at Lund University for his great support and encouragement of my research, and to my examiner Sven Robertz for his helpful input and feedback.

Contents

1 Introduction **7**

1.1 Research Objective . . . 8

1.1.1 Contribution to Research . . . 8

1.1.2 Research Questions . . . 8

1.2 Related Work . . . 8

1.2.1 CHERIoT . . . 9

2 Background **11**

2.1 Computer Architecture and Organization . . . 11

2.1.1 Instruction Set Architecture . . . 11

2.1.2 Instruction stalls . . . 12

2.1.3 Clocking Methodology . . . 13

2.1.4 Memory Hierarchy . . . 13

2.2 Memory safety . . . 14

2.3 CHERI . . . 14

2.3.1 Design goals . . . 14

2.3.2 CHERI Capabilities . . . 15

2.4 Next-Generation Radio Access Network . . . 16

2.5 Setup . . . 16

2.5.1 ARM Morello CPU . . . 16

2.5.2 LLVM-Morello . . . 17

2.6 Early performance results from the prototype Morello microarchitecture . . 18

2.6.1 Background . . . 18

2.6.2 Micro-architectural limitations . . . 19

3 Methodology **21**

3.1 Porting the software to the Morello hardware . . . 21

3.2 Performance Evaluation . . . 22

4 Implementation **23**
4.1 Porting the AArch64-ABI 23
4.2 Porting the Purecap-ABI 24
4.2.1 Heap Allocation in Pure-Capability Mode 24
4.2.2 Type-cast inherited bounds 25

5 Performance Testing and Evaluation **29**
5.1 Performance Results 29

6 Discussion **33**
6.1 Portability 33
6.2 Performance 33
6.2.1 Analysis 34

7 Conclusion **35**
7.1 Future work 35
7.2 Research Questions 36
7.3 Summary 36

References **37**

Appendix A Compiling a simple CHERI-C program using clang-morello C/C++ **43**

Chapter 1

Introduction

In a presentation at the 2019 BlueHat conference, Matt Miller, speaking on behalf of Microsoft presented an analysis of common vulnerabilities and exposures in their software and services. The report showed that around 70% of the vulnerabilities addressed through a security update each year are related to memory safety issues[18]. The main root causes were heap corruptions, use-after-free, type-confusions and uninitialized use-related vulnerabilities, also showing that spatial safety issues were the most pervasive vulnerability category[18]. Four years later in December of 2023, the National Security Agency (NSA) with its partners released "The Case for Memory Safe Roadmaps" Cybersecurity Information Sheet (CSI)", stating that *"The authoring agencies recommend software manufacturers evaluate multiple MSLs before integrating them into their programs of work"*[1], MSL being an abbreviation for memory-safe languages. These warnings show authorities' clear stance on memory-safety, the prevalence of memory-safety vulnerabilities in the software ecosystem and the importance for companies to take these vulnerabilities seriously.

There are several programming languages that solve the problems of memory-safety. Languages Like Java, C# and Go are instances of inherently memory-safe languages[19]. In addition, Rust has emerged in recent years as a fast, memory-safe language. By utilizing the ownership-based resource management design, an owner is introduced to each value which prevents memory-safety issues such as dangling pointers, buffer overflow/over-reads and uninitialized memory accesses[26]. Regardless, C/C++ are still some of the most widely used programming languages in the IT industry[17], meaning that these issues are unlikely to go away any time soon.

Over the past decade, researchers from SRI International and the University of Cambridge have been researching and developing the Capability Hardware Enhanced RISC Instructions (CHERI) architecture as a proposed solution to memory-safety related attacks for historically memory-unsafe languages such as the C/C++ programming languages[20]. With the launch of the Arm Morello CPU, companies can now implement and evaluate the performance of software on physical CHERI-supported hardware. Should existing applications be viably portable to CHERI while achieving satisfactory performance it could potentially have a significant impact on the global software ecosystem for C/C++ applications that would otherwise be conceivable victims of exploitable memory-safety vulnerabilities.

1.1 Research Objective

The aim of this master's thesis is to investigate the portability and performance of CHERI by applying it to 5th generation telecommunications software. In order to limit the scope of this thesis, the program is assumed to achieve memory-safety directly by compiling the program with pure-capabilities, relying on the *Formal proofs of the Morello security properties*[11]. Any required manual change was documented and explained. The portability metric was assessed by evaluating the required technical effort to retrofit the 5G applications to be run on the Morello board. More specifically, how much in the source code must be added or changed for the program to properly run. The performance is determined by running the program on a benchmark test and analyzing completed instructions, the cycle count and performance events through the built-in Arm performance counters.

1.1.1 Contribution to Research

Since the CHERI architecture can achieve memory safety for C/C++ applications without having to re-write the entire code base to a different language, it is a particularly attractive prospect for developers looking to improve software security. This thesis aims to serve as a practical example of how an industry application can be ported and run on the memory-safe CHERI architecture. Another desired outcome is to provide companies, researchers and other entities interested in this technology with valuable insights into how CHERI-compiled C/C++ based programs perform in relation to a conventional C/C++-compiled program.

1.1.2 Research Questions

1. Can 5G software be ported and run on the CHERI architecture?
2. How does the CHERI architecture affect the performance of critical base-station software?
3. How does the performance of the ported pure-capability 5G software compare against other existing CHERI benchmarks?

1.2 Related Work

Although there are several scientific reports on the performance and security around CHERI, there has been limited published research on how the architecture works in practice for software applications in the industry.

1.2.1 CHERIoT

CHERIoT: Complete Memory Safety for Embedded Devices is an article published in collaboration with and the CHERI research team at the University of Cambridge as well as researchers from ARM, Microsoft and Google[2]. The article presented an adaption of the CHERI-architecture for embedded systems and evaluated it on an end-to-end IoT application. The report demonstrates that an fine-grained memory protection safety can be enforced with a *capability-based ISA extension, co-designed with an RTOS and compartmentalization model*[2]. The report also show that strong security guarantees can be provided with modest hardware cost, and presents hardware extensions designed to improve the performance of a safe, shared heap.

Chapter 2
Background

2.1 Computer Architecture and Organization

A computer implementation can be divided up into two parts: organization and hardware[12]. Organization, also called the microarchitecture, defines high-level components of a computer such as the internal design of the CPU and the memory system. Hardware, on the other hand refers to detailed the logic design and packaging technology[12], such as integrated circuits.

2.1.1 Instruction Set Architecture

Instruction Set Architecture (ISA) is a component in the computer that can be viewed by the programmer. The principles of ISA can be understood by studying the Reduced Instruction Set Computer (RISC) and the five stages necessary to execute an instruction[12]:

Instruction Fetch (IF) fetches the next instruction from memory.

Instruction Decode (ID) decodes the instruction fetched. If the instruction is based on a condition, this stage tests the condition to see if a branch operation should be performed.

Execution (EX) executes the instruction using the Arithmetic Logical Unit (ALU). In this stage, arithmetic expressions, either on data in registers or on immediate, sign-extended instructions.

Memory Access (MEM) can read and write to memory. For a load instruction, the memory fetches data from the desired memory address.

Write-back (WB) writes the result of the instruction operation to the register specified in the instruction.

To maximize the use of these stages, the instructions are pipelined, meaning that as soon as an instruction goes from the IF stage to the ID stage, the IF stage can accomodate a new instruction. Therefore, a five-stage instruction pipeline can execute a maximum of five instructions in one clock cycle[12].

2.1.2 Instruction stalls

In a perfectly pipelined CPU core, one instruction will be executed in each pipe stage and all resources will constantly be occupied with a task. However, there are situations where a stage is unable to perform useful work and is forced to wait one cycle or more. For this reason, a stall is introduced which means that the CPU waits until it can perform useful work again. The stalls that can appear in a CPU can be divided into two categories: frontend stalls and backend stalls. Frontend stalls concern the instruction fetch stage and the instruction decoder stage, whereas backend stalls concern the execution stage, memory stage and write-back stage. An example of how the frontend and backend stalls are divided is displayed in Figure 2.1.

A branch-misprediction is one example of a frontend stall. This happens for instance when a superscalar CPU expects a logical if-statement to be true and starts fetching instructions from that branch. If the logical statement turns out to be false, all the fetched instructions have to be disregarded and new instructions have to be fetched, leading to wasted clock-cycles and additional stalls. The backend stalls concerns data-related stalls. When cache-misses occur, the CPU has to wait for the cache to refill the new data, and if other instructions depend on the fetched data they too have to stall. Backend stalls are also defined as each stalled CPU cycle where no micro-operations can be issued due to the fact that no micro-operations can write its results nor complete its execution[13]. Instructions that have reached the end of their pipeline stage in this way are described as "retired".

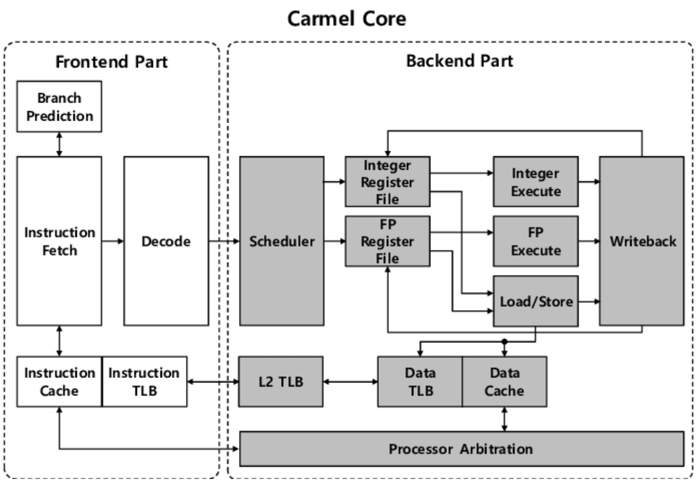


Figure 2.1: A representation of the Front and Backend of an Arm v8.2 Carmel CPU core. Image taken from the report *Memory-Aware Fair-Share Scheduling for Improved Performance Isolation in the Linux Kernel* [link]

2.1.3 Clocking Methodology

Reading and writing of data has to happen in a predictable, structured way in order to avoid undefined behavior. To create this predictability, there has to be a mechanism in the computer that describes when signals can be read and written. This is where the clocking methodology comes in: to determine when data is stable in relation to the clock and to ensure data validity[9]. The edge-triggered clocking methodology is one instance of such a mechanism, where stored values may only be updated at a clock edge, represented by an alternating control signal[9]. A visual representation of the edge-triggered clocking methodology is presented in Figure 2.2.

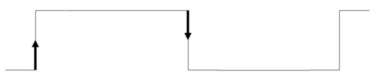


Figure 2.2: A representation of the edge-triggered clocking methodology.

2.1.4 Memory Hierarchy

In order to utilize resources efficiently, computer memory is divided up as a hierarchy. The top layer which lies closest to the Central Processing Unit (CPU) is small but fast, whereas each consecutive level becomes larger and slower in access time[4]. The main memory, or Random Access Memory (RAM) stores memory words in a binary representation called a cell, which can store 2^n elements and represent it as information[5]. The memory cells propagate over data buses, which are a number of physical wires. The RAM is volatile, meaning that the information will be lost once the power supply is disconnected, in contrast to non-volatile memory such as memory on the disk.

The memory unit that succeeds the RAM in a typical memory hierarchy is called a cache. The practical application of caches is based on two principles: Temporal locality, or the concept that memory which has been reached is likely to be referenced again in the near future, and spatial locality, the idea that memory which is physically close in proximity to the fetched memory is likely to be referenced in the near future[4]. Therefore, the cache memory is kept smaller than the main memory unit, thus requiring less clock cycles for loads and stores if the memory exists in the cache. Upon a cache miss when the desired information is not located, the cache fetches a line, which is a set of adjacent words, from the memory[4]. A computer usually consist of three caches, where the third level cache can be shared between multiple CPU:s.

In the top layer lie the special memory registers, which are responsible for direct communication with the CPU[22]. For instance, the Instruction Register (IR) holds the current instruction being executed, while the Program Counter Register (PC) holds the address of the instruction[22]. These registers are typically small, but data is also very fast to access.

2.2 Memory safety

A program has the property of memory safety when it ensures that every memory access is well defined as per the specification of the language[19]. This property is violated when memory accesses either happen to access memory locations that have not yet been allocated or having been freed which is called temporal safety violation, or when memory is accessed beyond the memory that has been specifically allocated for the program, called a spatial safety violation[19]. Memory-safe programming languages such as Java, C# and Go enforce memory safety. However, the C/C++ languages do not provide any information about whether a specified memory region is safe to access, exacerbated by type casts between pointers and the conflation between arrays and pointers[19].

An instance of a spatial safety violation is heap overflow, when out-of-bound memory access is done as a result of insufficient bound checking for memory that has been allocated to the heap[14]. An example of a heap overflow safety-violation is presented in listing 2.1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char* buffer = (char*)malloc(10 * sizeof(char));
    strcpy(buffer, "Hello");
    // Attempt to copy more data into the buffer than its allocated
    size
    strcat(buffer, " world! This is a heap overflow violation.");
    printf("Buffer contents: %s\n", buffer);
    free(buffer);
}
```

Listing 2.1: An example of a heap overflow

2.3 CHERI

2.3.1 Design goals

The University of Cambridge had the following design goals as a framework for what would become the CHERI architecture: Fine-grained memory protection, software compartmentalization, formal modeling and verification and a viable transition path. A short explanation of the design goals follow[24]:

Fine-grained memory protection aims to improve software resilience by addressing vulnerabilities caused by low-level bugs, such as buffer overflows as well as control-flow and pointer corruption. The objective of the CHERI-memory protection is to maintain the integrity, provenance and monotonicity of pointers, meaning that pointers should be unforgeable and restricted only to access their designated memory space.

Software compartmentalization aims to encapsulate memory and software into isolated components to reduce the impact of security vulnerabilities. This is achieved through in-address-space protection.

Formal modeling and verification aims to improve existing CPU and software by the introduction of new formal methodologies and to extend existing ones.

A viable transition path aims to improve hardware portability to make sure that software applications can benefit from increased software security without requiring significant modifications to the source code.

2.3.2 CHERI Capabilities

CHERI introduces architectural capabilities to current Instruction-Set Architectures (ISA) to improve the memory safety of historically memory-unsafe programming languages[24]. This is made possible by increasing the size of conventional pointer types to include access bounds and permissions. A capability pointer will be double the size of a conventional pointer plus one additional bit for the tag, which means that a 64-bit integer pointer will result in a 128-bit capability pointer with an additional 1-bit as a validity tag[24]. The additional 1-bit validity tag is out-of-band, meaning that it is independent from the register data, and follows the capability between register and memory. This feature allows programming languages like C/C++ which are inherently memory unsafe to be extended with safe pointers. The contents of a capability are presented in Figure 2.3. In contrast to the regular pointer which solely contains an address space, the capability pointer also contains bits for permissions, the object type and permission bounds.

The permission bits describe the architecturally defined capability permissions such as whether the capability is enabled or prohibited to load or store data. The o-type, or object type, describes what object the capability is as well as whether the capability is sealed or not, while the 32-bit bound is the address space that can be accessed. Sealed capabilities are immutable, non-dereferenceable capabilities that become invalid when modified.[24]. In order to accommodate this increased data width, the architecture extends General Purpose Registers (GPRs) to hold the increased size of the pointer as well as the tag bit[24]. Special-purpose registers are also extended. The program counter (PC) is also extended with bounds and permissions, called the Program Counter Capability (PCC).

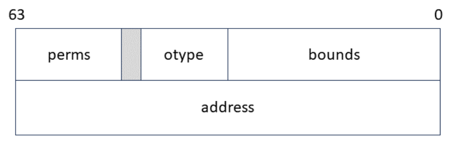


Figure 2.3: A figure depicting a 128-bit capability. The permission mask is parameterized, while the o-type and bounds are 18-bits and 32-bits respectively.

2.4 Next-Generation Radio Access Network

The 5th generation mobile network protocol as defined by the 3rd Generation Partnership Project (3GPP) is designed to standardize the control and management of cellular network radio resources. The User Equipment (UE) connects to a base station (gNodeB), which in turn is connected to the core network[21]. The connection, or link between the base station and the UE is called the bearer, and a UE can be simultaneously connected to multiple bearers at once. The protocol stack defines each layer necessary to connect and maintain the signal between the GnodeB and the UE. The protocols stack consist of the following: The Radio Link Control/Medium Access Control layer (RLC/MAC), the Packet Data Convergence Protocol (PDCP), the Radio Resource Control (RRC), and the physical layer. A brief description of each layer is presented below[21][10], and a visual representation in Figure 2.4.

Physical is responsible for channel encoding such as multiple-input, multiple-output (MIMO) and multiantenna processing as well as signal to time-frequency resource allocation.

Radio Link Control manages Service Data Units (SDUs) that are being sent from the PDCP, through SDU segmentation and re-transmission should the packets arrive corrupted. The frames that then ingress from the RLC to the MAC layer are called Package Data Units (PDUs). There are three transmission modes for transmission of upper layer PDUs: transparent mode (TM), unacknowledged mode (UM) and acknowledged mode (AM).

Medium Access Control handles the mapping between the logical and transport channels in order to find logical channels to the RLC and upper layers. It also handles scheduling and priority handling as well as error correction (ARQ and HARQ).

Packet Data Convergence Protocol manages the encryption/decryption and transfer of data between the SDAP layer and RLC. Each payload packet gets assigned a unique sequence number defined by the PDCP, which is then used for tracking successful delivery to the receiver. The sequence number is also used by the receiver to detect duplicate packages.

Service Data Adaption Protocol is responsible for the mapping of radio bearers and the Quality of Service (QoS) flows.

Radio Resource Control is responsible for the "dialogue" of the connection between the gNodeB and UE. RRC manages establishing, maintaining and interruption of the connection as well as broadcasting system information and participating in the handover of a connection based on the connection quality between the UE and the cells.

2.5 Setup

2.5.1 ARM Morello CPU

In collaboration with Linaro and the Universities of Cambridge and Edinburgh, Arm has developed the prototype system Morello. This experimental CPU is based on the Arm Neoverse-

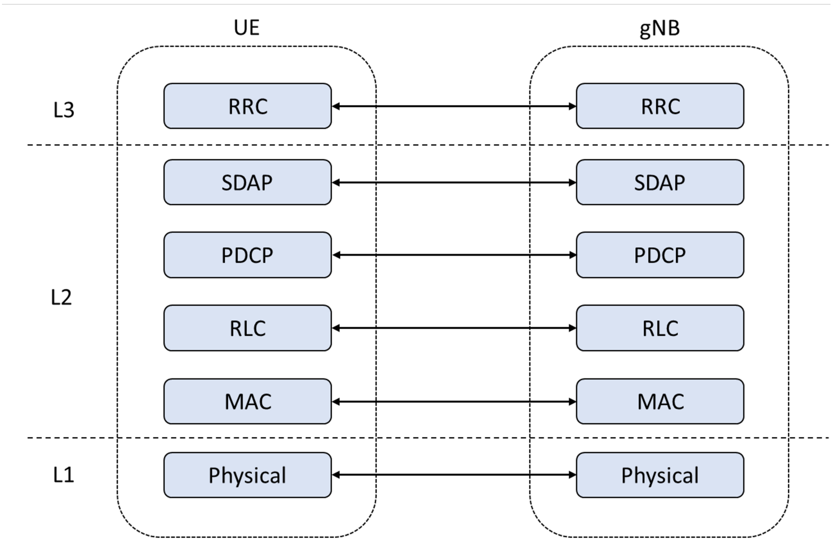


Figure 2.4: A representation of the 5G user-plane protocol stack

N1 CPU with a set of capability extensions. The hardware parameters for the modified Morello (SoC) processor are the same as the original: A 4-way decode, 8-way issue, 11 stage pipeline system which has a 128-entry re-order buffer and a three-level cache. The Morello CPU itself runs at 2.5GHz[11]. The operating system (OS) that was used was CHERIBSD OS version 14.0, which is a FreeBSD-modified OS designed specifically for CHERI.

The Morello CPU-Microarchitecture has been extended to accomodate the 128-bit capability registers. The capability registers are extended to hold the capability tag as well as the full capability data width, and the CPU caches have a meta-data tag every 128 bits[11]. The system buses are also expanded to carry the 1-bit capability tag alongside the capabilities. However, the data paths which connects to memory is not increased to accommodate the increased size of capability pointers, meaning that moving data between registers and memory take additional cycles[11]. There have also been changes to the load and store unit to ensure in-range memory accesses and that the capability has correct permissions.

The Morello architecture also defines a capability-based permission check. This check controls if a capability is prohibited from being stored to specific memory locations that restricts capability stores[11], and generates a memory fault if such is the case. The memory fault is used to enforce temporal safety for the Morello architecture[11], however support for temporal safety has not yet been fully integrated in the CHERIBSD version used for this thesis[25].

2.5.2 LLVM-Morello

LLVM is a project containing tools, libraries and header files to, as specified on the LLVM official website: *process intermediate representations and converts it into object files*[16]. One of the tools included under the LLVM umbrella is the Clang project, which for the C language family provides tooling infrastructure and a language front-end[15]. The Clang-project includes a ggc-supported compiler, which carries the same name as the project[15].

The CHERI LLVM-project is created particularly to support CHERI-compiled code, the version used for this project being the morello-compatible llvm version[8]. The LLVM-

Morello version particularly designed for Morello supports different compilation modes, such as native AArch64 C/C++ compilation, as well as the AArch64-PURECAP ABI, which assigns capabilites to all pointers. Note that AArch64 and AArch64-PURECAP ABIs cannot be linked together. A simple C-program compiled with LLVM-Morello is presented in Appendix A. LLVM-Morello was built using the cheribuild.py script defined by the University of Cambridge from their CTSRD-CHERI git repository[7]. The kernel file was generated on a Linux device, and then booted on the Morello CPU. The version that was selected was the CHERIBSD operating system with a Morello Pure-capability architecture, since it is the most mature OS out of the supported operating systems as of the writing of this report.

2.6 Early performance results from the prototype Morello microarchitecture

2.6.1 Background

In September 2023, the CHERI research team from the University of Cambridge released a technical report on how the SPECInt 2006 benchmark performed on the Arm Morello architecture using CHERI-capabilities[23]. The report also describes architectural limitations of the Morello microarchitecure. This report was used in the evaluation and analysis chapter for comparing and explaining the results produced in this thesis. The benchmark was first compiled with a hybrid AArch64 ABI, which was set as the baseline for the overhead calculations. Different capability-ABIs were then evaluated against this benchmark[23]. Descriptions of these compilation modes are presented below and the result of the benchmark tests in table 2.1.

Purecap ABI is the regular configuration. This mode uses capabilities for all pointers, including sub-language and and language-level pointers. These ABIs are also called AArch64c.

Benchmark ABI is a modified version of the Purecap ABI to combat certain shortcomings of the Morello architecture. The two main differences are that the global program counter capability (PCC) and the return capabilities are extended to have global bounds as a work around for the branch-predictor currently not supporting the prediction of bounds.

P128 and P128 Forced Global Offset Table (GOT) ABI are intended to analyze pointer-size growth and measure the associated overhead. Pointers and intptr_t are extended to the same width as the 128-bit capability while only the integer part is used by the code generated, meaning it can utilize existing AArch64 microarchitecture while enabling the analysis of the overhead of increasing the width of the pointer. Moreover, the compilation renders the access policy of global variables via the Global Access Table as per the default AArch64 policy.

Compilation mode	Overhead
Geomean Purecap ABI	28.01%
Geomean Benchmark ABI w/o data dependency fix (w/o larger store queue)	14.97%
Geomean Benchmark ABI w data dependency fix (w/o larger store queue)	7.40%
Geomean Benchmark ABI w data dependency fix (w/ larger store queue)	5.70%
Geomean P128 Forced GOT w data dependency fix (w/ larger store queue)	2.98%
Geomean P128 (w/ data-dependency fix) (w/ larger store queue)	1.82%

Table 2.1: Performance Results of the SPECInt Benchmark from the report *Early performance results from the prototype Morello microarchitecture*[23].

2.6.2 Micro-architectural limitations

There are four micro-architectural limitations of the Morello architecture that are mentioned in this report. These limitations have an impact on the performance of CHERI, but are believed to be resolvable[23]. The first limitation is the **PCC branch-prediction functionality**, where speculation of the changes in the PC bounds are prevented, resulting in additional stalls. The Benchmark ABI mentioned prior was created to solve this issue. Another limitation is the **Data-dependent exception**, which introduces a stall in the address translation of capability stores[23]. This stall is consciously introduced to handle data-dependent exceptions on stores. The third limitation revolves around **Untuned store throughput and buffer sizes**, which is caused by the memory bus not having been widened to accommodate store-pair instructions[23]. This results in store-pair 128-bit capabilities having to occupy two store-buffer entries instead of one. Lastly, there are limitations for **the MADD instruction** stemming from code-generation inefficiencies for some memory access on Morello.

Chapter 3

Methodology

This chapter describes the process of porting, running and evaluating the performance of 5G software onto the CHERI architecture using the Morello CPU. In order to narrow the scope of this thesis, only the RLC and MAC part of the 5G protocol stack was tested. The benchmark test is a pre-defined benchmark created to simulate packet flow and interaction between the RLC and MAC layer. The software was considered ported once all associated tests run successfully.

3.1 Porting the software to the Morello hardware

Software for the RLC-MAC functionality of the 5G OSI layer was added to the Morello board. The main objective was to make relevant changes in the code, and document the required changes that have to be made in order for the software to run. An RLC MAC benchmark test was used to determine whether the program runs properly or not. The code is written in C/C++, and was compiled using the LLVM-Morello clang compiler. Once the RLC and MAC benchmark passes the associated test, the program was copied to a separate version which will contain the pure-capability version of the program. Relevant changes were then made in order to run the program, and the changes were documented. The first version was then compared to the pure-capability version in terms of code required to correctly compile and run the associated test. The portability was evaluated by measuring the number of changed lines of C/C++ code that are required to be changed in relation to the total size of the code base.

3.2 Performance Evaluation

The performance evaluation was done by comparing how the regular-compiled version fares against the pure-capability version in terms of number of cycles and number of instructions executed. This comparison is selected to ensure equal CPU prerequisites, both for the architecture and the CPU itself. The AArch64 Performance Monitoring Unit (PMU) will then be used to calculate the clock cycles, execution time, number of retired instructions and to analyze performance events. The Morello architecture mostly follows the event space defined for an Armv8 implementation, but also has additional events related to capability instructions. When compiling a non-capability AArch64-ABI with LLVM-Morello, the capability-related events will show 0 as the program only executes Armv8 instructions. Furthermore, the program was locked to only run on one CPU in order to obtain more accurate results.

Two principal performance events were studied: Number of cycles and amount of instructions retired. To be able to access the PMU from user space, the PMUSERENR_EL0 flag has to be set for the program to function properly. Specifically for Morello, system permissions for userspace have to be set by the kernel. This can be done in cheribsd/sys/arm64/include/cherireg.h, where the flag CHERI_PERM_SYSTEM_REGS is included in the userspace permissions.

The performance events follow the ARMv8 implementation, with the addition of CHERI specific events in the address space 0x0200-0x03FF. The additional instructions that was of particular interests are listed below and are specified exactly as per the Arm® Architecture Reference Manual Supplement Morello[3]. These instruction are particularly interesting as the number of RD/WR capability instructions can be compared with the regular AArch64 ABI as to better understand the difference in performance.

L1D_CACHE_RD_CTAG (0x021C): Attributable Level 1 data cache access, read, valid capability. The counter counts each access counted by L1D_CACHE_RD which loaded a valid capability.

L1D_CACHE_WR_CTAG (0x021D): Attributable Level 1 data cache access, write, valid capability. The counter counts each access counted by L1D_CACHE_WR which stored a valid capability.

L2D_CACHE_RD_CTAG (0x0226): Attributable Level 2 data cache access, read, valid capability. The counter counts each access counted by L2D_CACHE_RD which loaded a valid capability.

L2D_CACHE_WR_CTAG (0x0227): Attributable Level 2 data cache access, write, valid capability. The counter counts each access counted by L2D_CACHE_WR which stored a valid capability.

Two key functionalities of the RLC and MAC software were studied: the receival of MAC Protocol Data Units (PDUs), and the construction of MAC PDUs, where each components will have separate measurement points. The RLC-MAC code was first compiled with regular AArch64 ABI with O3 optimization, and thereafter with AArch64-Purecap ABI O3. An evaluation was made on execution time of the benchmark as well as performance before and after compiler optimization. The compiled version of the AArch64-Purecap ABI was then be compared to how well it performs relative to the benchmark AArch64 ABI.

Chapter 4
Implementation

In this chapter, the porting and running of the RLC and MAC-5G components on the Morello architecture are described more in detail. The first part of the chapter describes the process of porting and running the RLC MAC software to Morello with the use of conventional clang compilation. Secondly, statistics over the required number of changes made is presented, followed by a more in-depth description of the changes required in order to compile and run the benchmark with CHERI capabilities.

4.1 Porting the AArch64-ABI

Two versions of the same benchmark test were created: One which generated a functioning AArch64 ABI and one a Purecap-ABI. In order to adapt the software to the CheriBSD operating system, the distinctions of what constitutes a change due to CHERI needed to be made. For instance: certain libraries required for the program to run have to be re-purposed for CheriBSD due to the program originally having been developed for Linux. A library not existing can lead to multiple changes having to be made in the code for the program to be built. One instance of this is the Linux Tracing Tool Next Generation (LTTNG). As of writing this report there is no known working version for the tool on CheriBSD, meaning that all trace-points in the original code have to be removed if the program is to be run. This change will be categorized as indirect, while all purely CHERI-related changes will be categorized as direct.

The direct and indirect changes made to the code are presented in Figure 4.1. These changes were extracted by breaking down the git diff between the unedited version and the complete version. Note that additions, deletions and changes of a line all constitute "changes", and that these changes are represented as a percentage of the lines changed compared to the total code base. The number of lines of C++ code made up 1.03% of the total lines of C++ in the program. Only indirect changes had to be made no further lines for the C++ code other than the ones changed for the AArch64 ABI compilation had to be made.

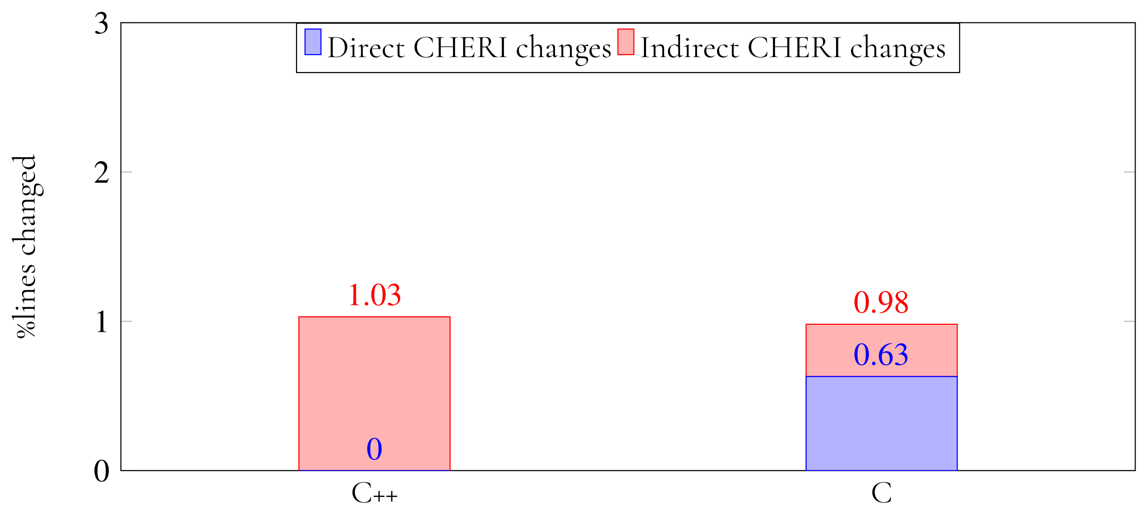


Figure 4.1: Changes required to port and run the CHERI-adapted RLC-MAC benchmark. Direct changes are all changes related to CHERI-capabilities, while the indirect changes related to working around libraries not yet ported to CHERI.

The indirect changes which required the most amount of effort came from having to decouple the LTTNG tool from the RLC-MAC software which is currently not supported on CheriBSD. Nevertheless, the software was not dependant on any key functionalities of LTTNG, meaning it could be disregarded. The second largest change had to do with name differences between include files for Linux and FreeBSD, which had to be updated accordingly. Additionally, the Google Protobuf library which is a mechanism that serializes structured data had to be ported in order to generate the required files. A ported version of this library could be found in the CHERI-PORTS-DASA section of the CHERI git.

4.2 Porting the Purecap-ABI

4.2.1 Heap Allocation in Pure-Capability Mode

The heap allocator used for the RLC-MAC software had to be modified in order to work with capabilities. Most notably, the size of objects like structs will increase in relation to regular C-programs if they contain capabilities. What this means is that if an original C or C++ program has been designed assuming a certain length of the pointers, certain changes have to be made to assure functional correctness and efficiency when porting a CHERI-version of the program. For a 64-bit architecture like AArch64-Morello Purecap, every capability increases the total size of the struct by 8, which can have implications for data alignment as described in Figure 4.2. One solution is to remove or decrease the size of sub-objects in the struct in order to compensate for the increased space that the capability pointer requires. Another solution is to increase the page size on the heap allocator at the cost of reducing the number of page entries. For this thesis, the former was chosen by reducing the maximum buffer capacity for a sub-object.

Another change had to do with the inheritance of capability bounds. When assigning a struct using memory derived from an object with a larger memory space, narrowing the bounds may be necessary if the struct only needs access to parts of the larger memory. This can be achieved using the cheri_bounds_set() function, which takes a pointer and a size and returns a new capability with the boundaries from the given capability + the specified size. Note that pointers that are initialized to NULL and later assigned to the existing address will not inherit the bounds of the larger object. In this case, cheri_address_set() has to be used, which allows the NULL pointer to receive the same bounds and address as an existing pointer.

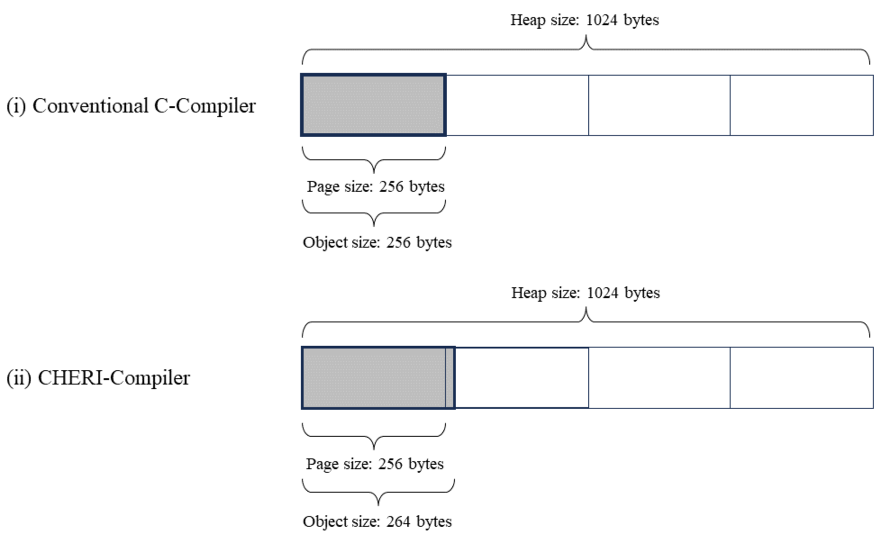


Figure 4.2: A figure depicting a theoretical heap allocator. The changed structural integrity of the object has to be accomodated for when using CHERI capabilities, otherwise the heap can be misaligned.

4.2.2 Type-cast inherited bounds

When porting the heap-allocator, a manual CHERI-adjusted change had to be made to ensure that each PDU was properly compartmentalized. A visualization of this problem is described in Figure 4.3. As a demonstration of the change, consider the following code in Listing 4.1. Here, a heap of size 1024 is created and a struct data_t that we want to insert into the heap. By assigning data_t into the first slot of the heap, the bounds of the heap object automatically gets inherited. Since the heap was written without regards to access bounds, this would give the object access to the entire heap, even with pure capabilities allowed. Figure 4.4 displays the results of executing Listing 4.1 without row 17.

One solution is to use the cheri_bounds_set(void *c, size_t x) function[25]. This function allows the programmer to narrow the bounds of a capability c so that the lower bound becomes the current address and size x determines the upper bound. With this function, the heap can be compartmentalized by narrowing the bounds of each object capability to only be able to access the bounds of their assigned slot. Figure 4.5 displays the results of executing listing 4.1 with row 17, which sets the bounds of "data" correctly.

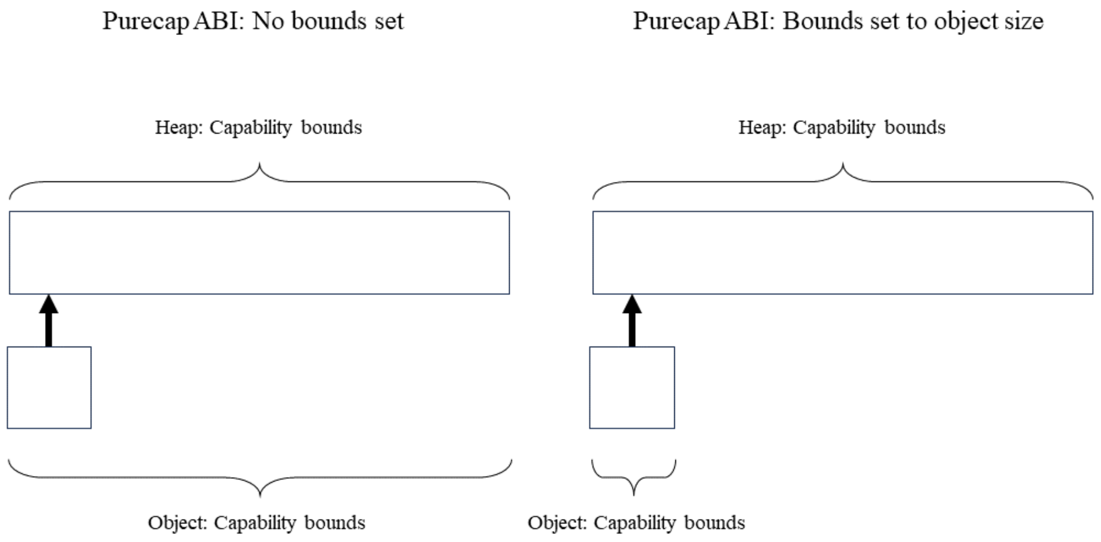
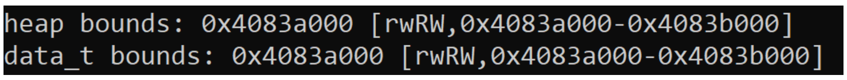
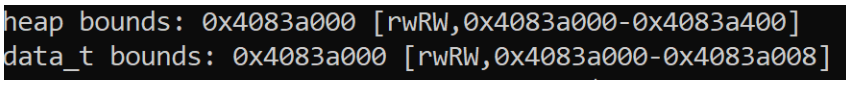

Purecap ABI: No bounds set
Purecap ABI: Bounds set to object size
Heap: Capability bounds
Heap: Capability bounds
Object: Capability bounds
Object: Capability bounds

Figure 4.3: A figure depicting a theoretical heap allocator. Type-cast inherited bounds may have to be narrowed manually to ensure full compartmentalization.



```
heap bounds: 0x4083a000 [rwRW,0x4083a000-0x4083b000]
data_t bounds: 0x4083a000 [rwRW,0x4083a000-0x4083b000]
```

Figure 4.4: A figure depicting the result of running the code described in Listing 4.1 excluding line 17. The bounds for the object data_t will inherit the bounds and permissions of the heap.



```
heap bounds: 0x4083a000 [rwRW,0x4083a000-0x4083a400]
data_t bounds: 0x4083a000 [rwRW,0x4083a000-0x4083a008]
```

Figure 4.5: A figure depicting the result of running the code described in Listing 4.1 including line 17. The bounds for the object data_t have been narrowed due to the cheri_bounds_set function.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cheriintrin.h>
4  #define HEAP_SIZE 1024
5
6  typedef struct data_t data_t;
7
8  struct data_t {
9     int x;
10    int y;
11 };
12 int main() {
13     int* heap;
14     data_t* data;
15     heap = (int*)malloc(HEAP_SIZE);
16     data = (data_t*)heap;
17     data = cheri_bounds_set(data,sizeof(data_t));
18     printf("heap bounds: %#p\n",heap);
19     printf("data_t bounds: %#p\n",data);
20 }
```

Listing 4.1: This code demonstrates how bounds can be narrowed after an object has inherited bounds from another object. The bounds for the capabilities "heap" and "data" would be the same without the cheri_bounds_set function written on line 17.

Chapter 5

Performance Testing and Evaluation

5.1 Performance Results

Table 5.1 and 5.2 presents the median of each event category after running the RLC MAC benchmark test. The values are presented as a percentage of how the pure-capability CHERI benchmark performed against the non-CHERI compiled benchmark. The difference between the number of retired instructions was relatively small, while the number of cycles were significantly larger. This results in an increased CPI, slowing down the total execution time. Figure 5.1 illustrates the overhead based on the execution time for both functions. The increase show that the Build Mac PDU function requires 53% more cycles in the version with pure-capabilites, which is 25% above the geometric mean of the performance for the SPECInt benchmarks.

Figure 5.2 show the increased number of data cache accesses for the pure-capability benchmark test. The largest increase came from the L1D cache writes in the Process Received MAC PDU function, which increased by 72% compared to the non-capability benchmark. This is in large contrast to the reads in the Process Received MAC PDU function, where the increase was only 36%. In addition to the cache reads and writes, table 5.2 presents the number of the read/write accesses where a capability tag was set in at least one part of the access. Despite the overall cache writes having increased the most for the Process Received Mac PDU, a larger portion of the Build Mac PDU writes had a capability tag set.

Event	Build Mac PDU	Process Received Mac PDU
CYCLES	53%	38%
INST_RETIRED	6%	17%
CPI	44%	17%
L1D_CACHE_RD	26%	36%
L1D_CACHE_WR	45%	72%
L2D_CACHE_RD	23%	19%
L2D_CACHE_WR	27%	24%
L3D_CACHE	18%	13%
STALL_FRONTEND	-33%	27%
STALL_BACKEND	122%	23%

Table 5.1: Performance Results of the RLC-MAC benchmark test

Event	Build Mac PDU	Process Received Mac PDU
L1D_CACHE_RD_CTAG	52%	58%
L1D_CACHE_WR_CTAG	55%	36%
L2D_CACHE_RD_CTAG	32%	37%
L2D_CACHE_WR_CTAG	31%	27%

Table 5.2: Shares of loads and stores where a capability tag was set in at least one part of the access

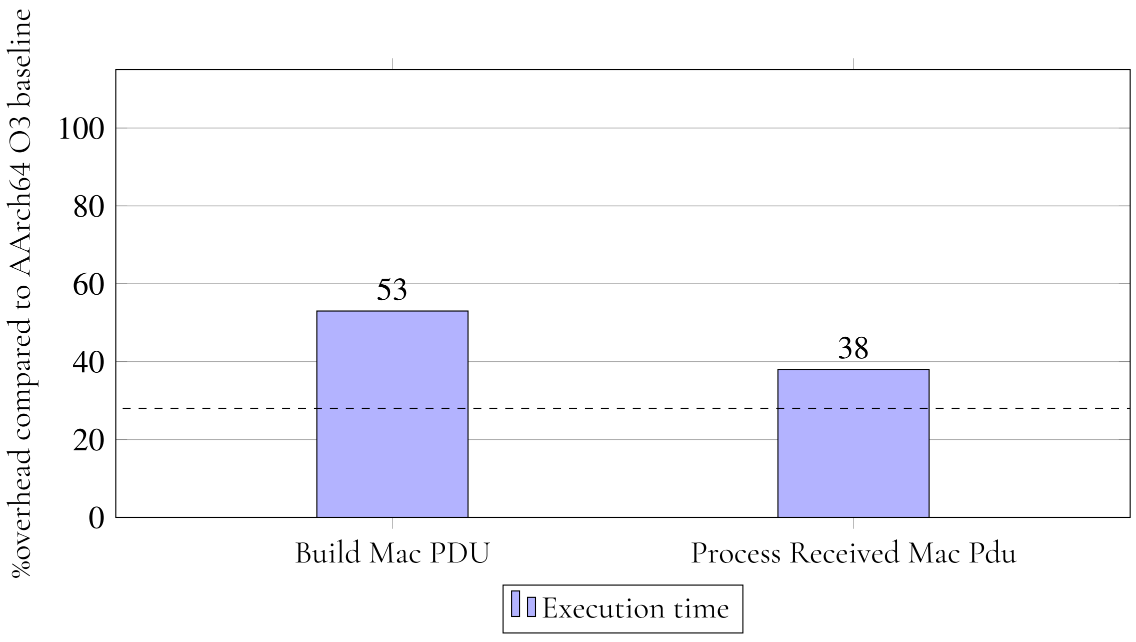


Figure 5.1: Overhead of the Purecap ABI compared to the baseline. The dashed line represents the geometric mean of the SPECint Purecap ABI.

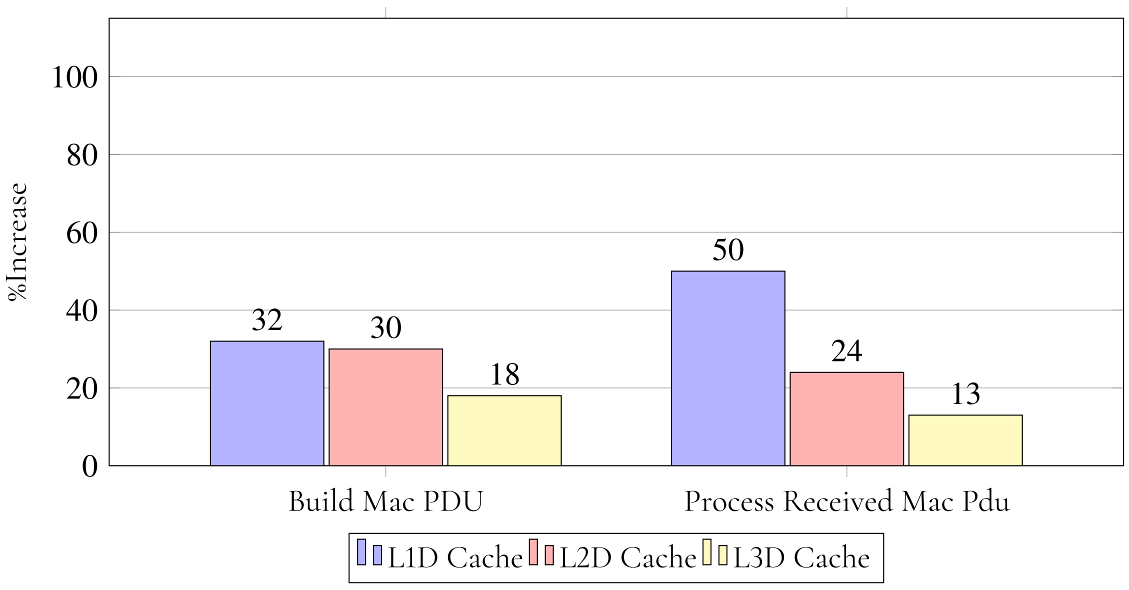


Figure 5.2: The increased number of data cache accesses for the Pure-cap ABI compared to the baseline

Chapter 6
Discussion

6.1 Portability

While parts of the code had to be adapted to a new operating system and to the Purecap ABI, the changes required were marginal in contrast to the massive code base that the 5G software component was built on. Allowing for the possibility of setting capability bounds significantly improved the flexibility of the C/C++ program and made it easy to tailor the bounds of pointers for specific purposes. Such was the case for the heap-allocator, were the bounds of capabilities could be narrowed to segment each element on the heap.

The libraries that were used could be found in the CHERI-ports-dasa section of the CTSRD-CHERI Github maintained by the University of Cambridge[6]. It is important to note that as of the time of writing this thesis, not all libraries are directly portable to CHERIBSD. Such was the case with LTTNG, which has a ported working version on FreeBSD which can be found under the FreeBSD equivalent of this port. On the other hand, a CHERI-compiled version of the googletest library did not require any changes to be ported onto CHERIBSD and compiled to a Purecap-ABI, meaning that there are libraries that the programmer can port themselves.

In regards to type-cast inherited bounds: It is important to manually check each case to make sure that the bounds are sufficiently restricted, as was the case for the RLC MAC Purecap-ABI: Even though the program passed all test cases, there were still pointers which arguably had access to more than was needed.

6.2 Performance

The performance of the measured functions Build Mac PDU and Process Received MAC PDU in the RLC-MAC benchmark had an overhead of 53% and 38% respectively compared to the baseline. The overhead could be derived from an increase in backend stalls, stemming from a increase in the number of reads and writes to the L1, L2 and L3 cache. Although this overhead was slightly higher than the SPECInt geometric mean of 28%, the results were anticipated and performed better than the 471.omnetpp and 483.xalancbmk SPECInt benchmarks.

The current version of Morello came with microarchitectural limitations presented in section 2.6.2. However, the results obtained from the SPECInt benchmarks also show a significant improvement in performance once these limitations are addressed. As an example, the Benchmark ABI with data-dependency fix and larger store queue decreased the overhead of the geometric mean from 28.01% to 5.70%. The Purecap-ABI that was compiled and evaluated in this thesis had no improvements to the microarchitecture, and so addressing the microarchitectural limitations of Morello could reduce the overhead of the Purecap-ABI implementation to similar levels.

6.2.1 Analysis

The overhead compared to the baseline can be attributed to the increase in number of cache accesses in all data caches, leading to an increase in the number of backend stalls. The microarchitectural limitations described in section 2.6.2 contribute to the increase in cache accesses. For instance, the data-dependent exception which adds a stall upon address translation of capabilities upon stores contributes to the increase in the number of backend stalls. Additionally, the store-pair instructions for 128-bit capabilities are executed in multiple cycles and occupy two store-buffer entries. These instructions are identified in an analysis of the generated assembly instructions for the Purecap ABI, were stp (store-pair) assembly instructions are used frequently.

Another contributing factor could be that the 128-bit capabilities require more memory than the 64-bit pointers, which would result in less information available at one time in the smaller cache and thus more cache misses overall. However, this does not necessarily explain the increase in L3 cache accesses for the Purecap ABI, since the cache size is large enough that it should be able to accommodate the increased size of a capability as opposed to a pointer. One hypothesis is that the increase could come from the size of objects such as structs having been chosen to perfectly align with cache blocks, whereas the increased size of capabilities would create a misalignment.

Despite the significant overall increase in backend stalls for the Build Mac PDU function, the frontend stalls decreased by 33%. One possible explanation is that possible frontend stalls have been masked by backend stalls, making it an artificial decrease. The frontend stalls for the Process Received MAC PDU function was significantly higher, and could depend on the PCC branch-prediction section 2.6.2.

Chapter 7
Conclusion

This thesis proves that complex, high-performing 5G software can be ported to the CHERI architecture, requiring few changes to the written C/C++ code. The performance tests of CHERI-enabled software show a 38-53% increase in overhead, which is believed to largely stem from solvable architectural limitations of Morello. Based on these findings, the conclusion is made that CHERI is a viable option for improving the memory-safety of C/C++ programs.

7.1 Future work

This thesis primarily focused on proof-of-concept of the CHERI architecture on 5G applications, whereas the performance was presented on the as-is implementation of the ported and run program. Future research should be focused on better understanding how the overhead can be mitigated, particularly by addressing the architectural limitations of Morello. A future Morello update could see the introduction of the Benchmark ABI, which would be ideal for evaluating how the 5G-software performs when these limitations are addressed.

Due to the limited scope of this thesis, only the RLC and MAC components of the OSI layer were studied. More comprehensive, all-encompassing simulations should be made to better understanding how CHERI-capabilities affect the correspondence between multiple components of the OSI layer. Another component that should be further understood is how temporal safety plays into the software, as this thesis mainly focuses on implementing spatial safety programming.

7.2 Research Questions

RQ1 Can 5G software be ported and run on the CHERI architecture? Yes. This thesis proved that the RLC and MAC functionalities can be ported and run with CHERI using the Arm Morello CPU, both for a compiled AArch64 ABI and a CHERI Purecap ABI. For the RLC and MAC program, only around 1% of the existing C/C++ code had to be rewritten.

RQ2 How does the CHERI architecture affect the performance of critical base-station software? The median overhead of the two most important functions in the RLC and MAC program was 53% and 38% for the CHERI Purecap ABI in relation to the AArch64 ABI.

RQ3 How does the performance of the ported pure-capability 5G software compare against other existing CHERI benchmarks? The overhead measured in the RLC and MAC benchmark test was compared to a SPECInt 2006 benchmark test of the Morello CPU conducted by researchers at the University of Cambridge. The geometric mean overhead of this benchmark was lower than the RLC and MAC, except for the 471.omnetpp and 483.xalancbmk benchmarks where the overhead was higher.

7.3 Summary

The results of this thesis show that the RLC-MAC component of the 5G layer can be ported and run on the Arm Morello architecture, requiring very few changes to the C/C++ code which the software is built on. In total, 1.03% of the total C++ code and 0.98% of the total C code had to be changed to fully port the CHERI-compiled version. The most significant CHERI-related changes were done in the heap allocator. The changes included alignment corrections such as reducing the size of the objects in order to accommodate the increased size of each pointer. Additionally, the bounds for some capabilities had to be manually narrowed in order to ensure full compartmentalization of the heap in the implementation.

The performance results show that the median overhead for the two measured functions were 53% and 38% respectively, stemming from an increase in L1,L2 and L3 data cache accesses. The increase in cache-accesses can be attributed to the capabilities requiring more memory in the cache, resulting in more cache-misses and in-term more clock cycles. Micro-architectural limitations of the Morello CPU are also factors that contribute to the increased overhead. One limitation described by the CHERI Research team at the University of Cambridge is that memory buses have not been widened, meaning that store-pair instructions executed on capability-pointers require additional cycles[23].

In comparison to the Geometric mean of the Purecap ABI in the SPECInt 2006 benchmark which was tested on the Morello CPU by researchers at the University of Cambridge, the measured overhead from the performance test of the 5G software was higher. However, the performance was still lower than certain individual benchmarks, and the overhead is expected to drop once architectural limitations of the Morello CPU are addressed.

References

[1] National Security Agency. The Case for Memory Safe Roadmaps: Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously. Online publication, December 2023. Accessed: April 22, 2024.

[2] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 641–653, New York, NY, USA, 2023. Association for Computing Machinery.

[3] Arm Limited. *Arm Architecture Reference Manual Supplement - Morello for A-profile Architecture.* Arm Limited, 2022. Document Number DDI0606 Version A.k.

[4] Gérard Blanchet and Bertrand Dupouy. *Memory*, chapter 7, pages 139–156. John Wiley Sons, Ltd, 2012.

[5] Gérard Blanchet and Bertrand Dupouy. *The Basic Modules*, chapter 2, pages 17–34. John Wiley Sons, Ltd, 2012.

[6] CTSRD-CHERI. cheribsd-ports-dasa. `https://github.com/CTSRD-CHERI/cheribsd-ports-dasa`. Accessed: 2024-05-01.

[7] CTSRD-CHERI. cheribuild.py - a script to build cheri-related software (requires python 3.6+). `https://github.com/CTSRD-CHERI/cheribuild?tab=License-1-ov-file#readme`. Accessed: 2024-04-10.

[8] CTSRD-CHERI. The CHERI LLVM Compiler Infrastructure. `https://github.com/CTSRD-CHERI/llvm-project/blob/master/README.md`. Accessed: 2024-04-10.

[9] Patterson David A. and Hennessy John L. *Computer Organization and Design RISC-V Edition : The Hardware Software Interface.*, volume [First edition] of *ISSN*. Morgan Kaufmann, 2018.

[10] José Luiz Frauendorf. *The Architectural and Technological Revolution of 5G*. Springer International Publishing, 2023.

[11] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro*, 43(3):50–57, 2023.

[12] John L. Hennessy, David A. Patterson, and Krste Asanović. *Computer architecture. a quantitative approach*. Morgan Kaufmann/Elsevier, 2012.

[13] Jungho Kim, Philkyu Shin, Myungsun Kim, and Seongsoo Hong. Memory-aware fair-share scheduling for improved performance isolation in the linux kernel. *IEEE Access*, PP:1–1, 05 2020.

[14] Ashish Kundu and Elisa Bertino. A New Class of Buffer Overflow Attacks. *2011 31st International Conference on Distributed Computing Systems, Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 730 – 739, 2011.

[15] LLVM. Clang: a c language family frontend for llvm. `https://clang.llvm.org/`. Accessed: 2024-05-17.

[16] LLVM. Getting started with the llvm system. `https://llvm.org/docs/GettingStarted.html#Overview`. Accessed: 2024-05-17.

[17] Dongdong Lu, Jie Wu, Yongxiang Sheng, Peng Liu, and Mengmeng Yang. Analysis of the popularity of programming languages in open source software communities. *2020 International Conference on Big Data and Social Sciences (ICBDSS), Big Data and Social Sciences (ICBDSS), 2020 International Conference on, ICBDSS*, pages 111 – 114, 2020.

[18] M. Miller. BlueHat IL 2019 - Matt Miller -Trends, Challenges, and Strategic Shifts. Online video, 2019. Accessed: 2024-05-04.

[19] S. Nagarakatte. Full Spatial and Temporal Memory Safety for C. *IEEE Security and Privacy*, pages 2–11 – 11, 2024.

[20] Department of Science and University of Cambridge Technology. Capability hardware enhanced risc instructions (cheri). `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`. Accessed: 2024-03-05.

[21] Wim Rouwet. *Open Radio Access Network (o-RAN) Systems Architecture and Design. [Elektronisk resurs]*. Elsevier Science amp; Technology, 2022.

[22] Shuangbao Paul Wang. *Computer Architecture and Organization. Fundamentals and Architecture Security*. Springer Nature Singapore, 2021.

[23] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. Early performance results from the prototype Morello microarchitecture. Technical Report UCAM-CL-TR-986, University of Cambridge, Computer Laboratory, September 2023.

[24] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023.

[25] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical Report UCAM-CL-TR-947, Computer Laboratory, June 2020.

[26] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. 2020. URL: `https://arxiv.org/pdf/2003.03296`.

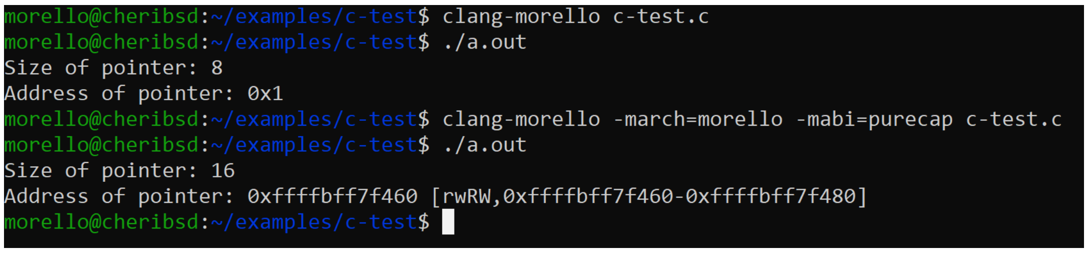
Appendices

Appendix A

Compiling a simple CHERI-C program using clang-morello C/C++

Compiling, linking and running C and C++ programs works like any regular program compiled with clang. Only when the compiler flags -march=morello and -mabi=purecap are included in CFLAGS and LDFLAGS will the program be compiled with CHERI capabilities. Consider the basic C-program c-test.c with an int pointer *a*, where the size and address of *a* is printed. In the default case, the size of a will be 8 bytes, whereas the address will be 0x01. However, compiling the program with the Purecap flags returns *a* pointer of size 16, and an address space for which memory addresses the pointer is allowed to access. Trying to access memory addresses outside of this range with the capability will throw an exception.

Listing A.2 show standard AArch64-generated assembly code while Listing A.3 show the Purecap equivalent, both compiled from the same C code in Listing A.1. Added assembly instructions such as scbnds and clrperm are necessary to modify the boundaries and permissions of the capability. The results of the compilation are shown in Figure A.1. In the figure, the address space was printed using #p, which is a cheri-specific instruction to allow the programmer to study the bounds and permissions of a capability. Additionally, if the programmer wants to edit the capabilities themselves they should use the *cheriintrin.h* library, which contains important functions for narrowing and aligning bounds.



```
morello@cheribsd:~/examples/c-test$ clang-morello c-test.c
morello@cheribsd:~/examples/c-test$ ./a.out
Size of pointer: 8
Address of pointer: 0x1
morello@cheribsd:~/examples/c-test$ clang-morello -march=morello -mabi=purecap c-test.c
morello@cheribsd:~/examples/c-test$ ./a.out
Size of pointer: 16
Address of pointer: 0xffffbff7f460 [rwRW,0xffffbff7f460-0xffffbff7f480]
morello@cheribsd:~/examples/c-test$ 
```

Figure A.1: Compiling and running the code in Listing A.1, first to a normal AArch64-ABI and then to a Purecap-ABI.

```c
    #include <stdio.h>

    #ifdef __CHERI_PURE_CAPABILITY__
    #define PRINTF_PTR "#p"
    #else
    #define PRINTF_PTR "p"
    #endif

    int main(void){
        int* a;
        printf("Size of pointer: %zu\n", sizeof(a));
        printf("Address of pointer: %"PRINTF_PTR"\n", a);

    }
```

Listing A.1: C Example

```
0000000000010a3c <main>:
   10a3c: ff 03 81 02    sub     csp, csp, #64             // =64
   10a40: fd 7b 81 42    stp     c29, c30, [csp, #32]
   10a44: fd 83 00 02    add     c29, csp, #32             // =32
   10a48: e0 d3 c1 c2    mov     c0, csp
   10a4c: 08 02 80 52    mov     w8, #16
   10a50: 08 00 00 f9    str     x8, [c0]
   10a54: 00 38 c8 c2    scbnds  c0, c0, #16               // =16
   10a58: 09 70 c6 c2    clrperm c9, c0, wx
   10a5c: 80 00 80 90    adrp    c0, 0x20000 <main+0x60>
   10a60: 00 c4 42 c2    ldr     c0, [c0, #2832]
   10a64: 23 00 00 94    bl      0x10af0 <printf+0x10af0>
   10a68: e1 07 40 c2    ldr     c1, [csp, #16]
   10a6c: e0 d3 c1 c2    mov     c0, csp
   10a70: 01 00 00 c2    str     c1, [c0, #0]
   10a74: 00 38 c8 c2    scbnds  c0, c0, #16               // =16
   10a78: 09 70 c6 c2    clrperm c9, c0, wx
   10a7c: 80 00 80 90    adrp    c0, 0x20000 <printf+0x10abc>
   10a80: 00 c8 42 c2    ldr     c0, [c0, #2848]
   10a84: 1b 00 00 94    bl      0x10af0 <printf+0x10af0>
   10a88: e0 03 1f 2a    mov     w0, wzr
   10a8c: fd 7b c1 42    ldp     c29, c30, [csp, #32]
   10a90: ff 03 01 02    add     csp, csp, #64             // =64
   10a94: c0 53 c2 c2    ret     c30
```

Listing A.2: AArch64-Purecap assembly

```
000000000210b3c <main>:
  210b3c: ff 83 00 d1    sub     sp, sp, #32               // =32
  210b40: fd 7b 01 a9    stp     x29, x30, [sp, #16]
  210b44: fd 43 00 91    add     x29, sp, #16              // =16
  210b48: 80 ff ff 90    adrp    x0, 0x200000 <register_classes>
  210b4c: 00 c0 15 91    add     x0, x0, #1392             // =1392
  210b50: 01 01 80 d2    mov     x1, #8
  210b54: 33 00 00 94    bl      0x210c20 <printf@plt>
  210b58: e1 07 40 f9    ldr     x1, [sp, #8]
  210b5c: 80 ff ff 90    adrp    x0, 0x200000 <register_classes+0x14
    >
  210b60: 00 18 16 91    add     x0, x0, #1414             // =1414
```

```
210b64: 2f 00 00 94    bl       0x210c20 <printf@plt>
210b68: e0 03 1f 2a    mov      w0, wzr
210b6c: fd 7b 41 a9    ldp      x29, x30, [sp, #16]
210b70: ff 83 00 91    add      sp, sp, #32             // =32
210b74: c0 03 5f d6    ret
```

Listing A.3: AArch64 assembly

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-05

EXAMENSARBETE Memory-Safe 5G Software Using the CHERI Hardware Architecture
STUDENT Fredrik Hessner
SUPERVISOR Jonas Skeppstedt (LTH) Håkan Englund (Ericsson)
EXAMINER Sven Robertz (LTH)

Memory-safe 5G software with the CHERI architecture

POPULAR SCIENCE SUMMARY **Fredrik Hessner**

Improving memory-safety for inherently unsafe C/C++ applications is very important, but has to be made scalable without compromising too much on performance in order to be feasible. This thesis demonstrates that 5G software can be ported and run on the memory-safe CHERI-architecture.

Changing your home address typically involves contacting the authorities and registering your new address with them. Imagine if you instead could maliciously register yourself on your neighbors address and with that gain full permission to enter and make changes to their home. This example may sound a bit bizarre, but it holds similarities to memory safety vulnerabilities of memory-unsafe programming languages. This master's thesis explores Capability Hardware Enhanced RISC Instructions (CHERI) that aims to address these vulnerabilities.

In this thesis, 5G software was implemented on the CHERI architecture. The objective was to evaluate if it was possible to run part of the 5G functionalities on CHERI, and if so, how many changes to the code would be required. The hardware used was the Arm Neoverse N1 with Morello SoC, which is a processor that can compile computer code with CHERI-capabilties. These capabilities introduce concepts like permissions and access bounds for otherwise memory-unsafe programming languages, particularly C and C++. Once the software was ported, the CHERI implementation was tested against a non-CHERI version. This allowed us to measure the overhead that a CHERI-compiled version produced. The results were then compared with existing performance evaluations of the Morello CPU.

The results show that the CHERI-architecture is portable to the 5G software. In fact, only around 1% of the C/C++ code had to be rewritten for the program to run. The overhead was a few percentage points higher than the performance of existing results, but is likely to drastically decrease if architectural limitations of the hardware are addressed.

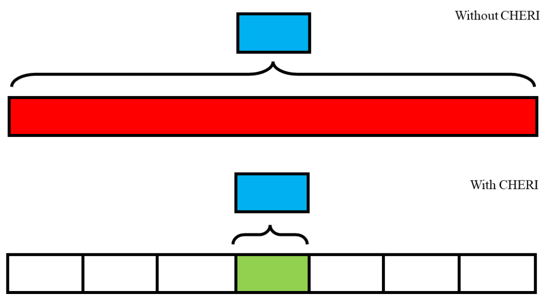


Figure 1: Compartmentalized memory addresses using CHERI.