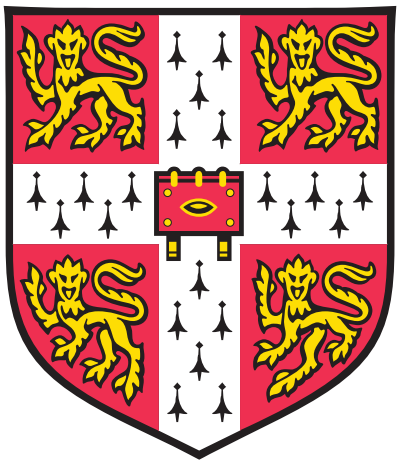
Verified security properties for the capability-enhanced CHERI-MIPS architecture

Kyndylan Nienhuis

The Computer Laboratory
University of Cambridge



September 2021, St Edmund's College
This thesis is submitted for the degree of
Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This thesis does not exceed the prescribed limit of 60 000 words.

Verified security properties for the capability-enhanced CHERI-MIPS architecture

Kyndylan Nienhuis

Abstract

Despite decades of research, the computer industry still struggles to build secure systems. The majority of security vulnerabilities are caused by a combination of two fundamental problems. First, mainstream engineering methods are not suited to find small bugs in corner cases. Second, mainstream hardware architectures and C/C++ language abstractions provide only coarse-grained memory protection, which lets these small bugs escalate to serious security vulnerabilities. CHERI, the context of our work, is an ongoing research project that addresses the second problem with hardware support for fine-grained memory protection and scalable software compartmentalisation. CHERI achieves this by extending commodity hardware architectures with a capability system, in which all memory accesses must be authorised by a capability. Capabilities are unforgeable tokens that contain address bounds and permissions, determining the memory region and the types of accesses they can authorise. They can be passed around and manipulated only in specific ways. CHERI has been initially developed as CHERI-MIPS, with later work on CHERI-RISC-V.

In this thesis, we address the first problem in the context of CHERI: we formally state and prove security properties for the CHERI-MIPS architecture. Our first set of security properties forms a new abstraction layer of CHERI-MIPS, explaining execution steps in terms of nine abstract actions, one for each kind of memory access, capability manipulation, and security domain transition. We use this abstraction to reason about arbitrary code: we characterise which capabilities can be accessed or constructed by potentially compromised or malicious code and which memory locations it can overwrite until it transitions to another security domain. We use this to prove the correctness of a simple compartmentalisation scenario, in which CHERI's capability system is used to isolate a component from the rest of the system. Our results are based on a full, non-idealised, sequential specification of CHERI-MIPS, which is complete enough to boot an operating system. The challenges of proving our properties include the size of the architecture, its easy-to-miss corner cases, and the fact that the architecture keeps evolving.

As a step towards CHERI's industrial adoption, the Morello program is developing the eponymous prototype CHERI extension of the Armv8-A architecture, along with a processor implementation, development board, and software. Building on our work, the formal verification of architectural security properties is an important part of this program.

Acknowledgements

First and foremost, I am grateful to my supervisor, Peter Sewell, for his excellent guidance throughout my PhD. He gave me the freedom to change course in the middle of my PhD and provided patient support when I was finishing up. His encouragement and advice were invaluable.

Many colleagues have helped me with various aspects of my research. I thank Thomas Tuerk and Dominic Mulligan for teaching me the intricate aspects of Isabelle; I thank Anthony Fox for providing an Isabelle/HOL export of L3 and answering many questions; I thank Alexandre Joannou for explaining his CHERI-MIPS specification; I thank Robert N. M. Watson, Simon W. Moore, Alexandre (again), Michael Roe, and Jonathan Woodruff for many interesting discussions about desired properties of CHERI and whether strange behaviours I found were bugs or features; and, finally, I thank Thomas Sewell and Thomas Bauereiss for providing useful feedback on my thesis draft. I also thank my examiners Larry Paulson and Deepak Garg for their useful feedback and interesting questions.

I consider myself lucky to have been at the Computer Laboratory at the same time as Kayvan Memarian, Jean Pichon-Pharabod, Justus Matthiesen, Thomas Bauereiss, David Kaloper-Meršinjak, Christopher Pulte, Ohad Kammar, Kasper Svendsen, Hannes Mehnert, and Dominic Mulligan. Thank you for all the enjoyable conversations, dinners, and board games. Through the Gates Cambridge Scholarship, I have met wonderful and inspiring people. Jessica, Farhan, Elijah, Annika, and many others, it was a privilege to be among you. My special thanks go to my dear friends Wicher and Michał, who have made my time in Cambridge truly unforgettable.

Throughout my life, I have had excellent teachers who sparked my interest in a wide variety of topics. In particular, I thank Rob Piet, whose mathematics lessons were never boring and who encouraged me to participate in the International Mathematical Olympiad; and I thank Roland van der Veen for his unstoppable enthusiasm when supervising my Bachelor thesis.

Last but not least, I am eternally grateful to Aad and Jacqueline for their unconditional love and support, to Joscelyn for being an incredible brother, and to Clara and our beloved Ewout who mean everything to me.

Funding

This work received funding from the following sources:

- the Gates Cambridge Trust,
- the Computer Laboratory,
- EPSRC programme grant EP/K008528/1 (REMS: Rigorous Engineering for Mainstream Systems),
- the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 789108, ELVER), and
- the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-18-C-7809 (CIFV). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Contents

Chapter 1

Introduction

Despite decades of research, the computer industry still struggles to build secure systems: personal devices can be compromised by just visiting a website, corporate IT systems are breached exposing intellectual property and customer data [142, 115], and even the highly defended systems of the National Security Agency [138] and Iran's uranium enrichment facilities [82] have been hacked. At the same time, the industry builds impressive systems that, for example, autonomously fly aircraft, beat the best human players at Go [141], and that allow us to communicate almost instantaneously with the other side of the world. Why are these feats possible, but secure computer systems remain elusive?

There are of course many reasons. Some companies do not have an incentive to develop a secure product because their customers are not willing to pay for security and they are not liable for vulnerabilities in their product. Furthermore, users make mistakes and can for example be tricked into revealing their passwords. But even if we focus on the technical side and assume honest, incentivised, and capable developers, it is difficult to build a secure system because of the following two fundamental reasons.

First, mainstream engineering methods are not suited to find small bugs in corner cases. Under normal use, these corner cases do not occur often, so these methods are sufficient to develop systems that work well enough under normal use. Attackers, however, can deliberately guide a system to these corner cases, triggering the bug whenever they want.

Second, mainstream hardware architectures and C/C++ language abstractions provide only coarse-grained memory protection, which means that too often these small bugs escalate to serious security vulnerabilities. For example, a buffer overflow is a simple bug that can corrupt memory. While accidental memory corruption typically crashes the program, an attacker can corrupt memory in specific ways to leak sensitive information or to gain full control over the program.

A classic example of the combination of the above two problems is the Heartbleed vulnerability [40] in OpenSSL [113]. The bug was introduced in 2012, and publicly disclosed

and patched in April 2014. One of the reasons the bug remained undiscovered for so long was that OpenSSL functioned correctly as long as clients would send correct messages to the server. However, when a client would send a specific malformed message, the server would leak its private memory, which could contain emails, passwords, private keys of certificates, and other sensitive information. Figure 1.1 on the following page illustrates the vulnerability.

Below we discuss the memory protection that mainstream hardware architectures provide and we explain why this could not prevent Heartbleed.

1.1 Protection through paged virtual memory

In the mid 1950s computers had two levels of memory: a limited amount of magnetic core memory and a larger amount of slower magnetic drum memory. Programmers had to manage these levels themselves, swapping data between them to optimise execution speed. In the years 1956–1961 *virtual memory* was invented [61, 52, 76], which greatly simplified this: the programmer only sees a large, virtual address space, and a management system is responsible for mapping this to physical memory and ensuring that frequently-used data is in the first level of memory. The management system, which can be a combination of hardware and of system software, typically achieves this by dividing the virtual address space into *pages*. When a page is first used it is *mapped* to the first level of physical memory. When this memory runs out of free space, the least used pages are moved to the second level of memory, and these pages are moved back if they are used again.

In the 1960s virtual memory was also used to protect memory. The Burroughs 5000 architecture divided the virtual address space into *segments* that each had an associated bit describing whether the segment contained code or data [86, 89]. Segments containing data could not be executed, segments containing code could not be read or written to, and only privileged code could change these bits. This protection mechanism fell out of use, but was revived in the early 2000s when worms such as Slammer [96] and Blaster [8] spread across the world. These worms infected their hosts by injecting code and letting the host execute it, which depends on memory being writable and executable. To prevent these attacks architectures implemented a similar protection mechanism, called *no-execute* in AMD64 [3, Volume 2, §5.4.1], *execute disable* in Intel 64 [71, Volume 3, §5.13.2], and *execute never* in Arm [5, B3.5.7].

Virtual memory can provide another form of protection, namely memory *isolation*. Operating systems (OSs) store the translation from virtual to physical addresses in a *page table*. They maintain a different page table for each process and ensure that these map to distinct physical addresses, which means a process cannot address physical memory that is in use by another process. This form of isolation is invaluable to isolate the memory of

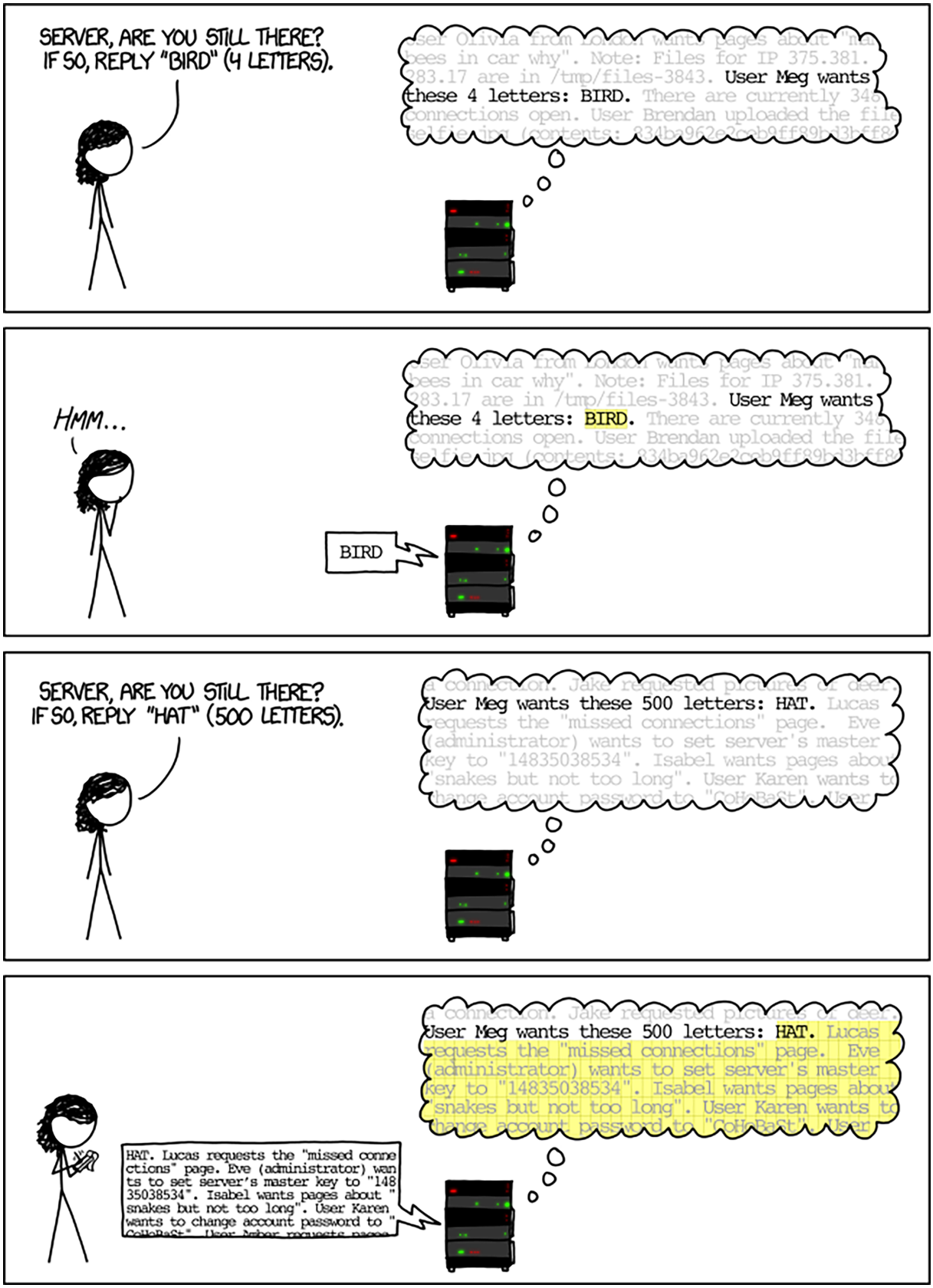


Figure 1.1: The Heartbleed vulnerability explained by xkcd [99].

different programs, but it can also be used to isolate different parts of the same program, which is called *software compartmentalisation*.

1.1.1 Software compartmentalisation

Software compartmentalisation follows the *principle of least privilege* [133] which says that each part of a program should run only with the permissions it needs to function. When visiting two websites in a browser, for example, one website does not need information that is entered in the other website, so by restricting its permissions one can ensure that even a compromised or malicious website cannot access this information. One can achieve this by rendering each website in its own process, which, for example, the Chromium browser does by default [10, 123].

Besides complete isolation, compartmentalisation can also be used to control the extent of interaction. Returning to the browser example, some websites need to access the file system but should not be able to do this arbitrarily. The Chromium browser realised this by creating a (trusted) process that can access the file system and that implements security policies. Renderer processes do not have direct access to the file system but can request access through the trusted process, which means even compromised websites are subjected to the security policies.

Unfortunately, compartmentalisation via processes does not scale well. The main issue is the limited size of the *translation lookaside buffer* (TLB). When an instruction accesses a virtual address, the hardware checks whether its translation is cached in the TLB. If it is not – a *TLB miss* – the translation is looked up in the page table, inserted in the TLB, and the instruction is restarted. This involves several memory accesses and delays the instruction by 10–100 cycles, but it does not significantly affect the overall performance as TLB misses are typically rare [129]. This changes when compartmentalising a program into a large number of processes. Because each process has its own address space, the working set of the program is fragmented over many pages and the TLB might no longer be large enough to cache the mappings of these pages. If this happens the TLB *thrashes*: handling a TLB miss causes a cached translation to be evicted, which soon leads to another TLB miss, et cetera, and the overall performance collapses.

Even if the TLB does not thrash there is a memory and performance overhead when compartmentalising via processes: the OS needs to allocate a page table for each process, partially used pages cannot be used by other processes, the OS has to save and restore process state when switching between processes, and most types of inter-process communication add their own overhead. The result is that this form of compartmentalisation mainly suits security critical programs that can be divided in a small number of relatively isolated compartments. It would be difficult to compartmentalise OpenSSL using processes while delivering the same performance.

1.2 The lack of memory safety in C/C++

Orthogonal to the memory protection that hardware architectures offer, some programming languages also offer memory protection. Unfortunately, many security critical programs such as OpenSSL are written in C or C++ which do not offer any memory protection. To understand why not, we relate their lack of memory safety to their original design goals.

C was invented in the years 1969–1973 as a system programming language. It was meant to compete with hand-written assembly in terms of execution speed and program size, while being easier to write and understand [126]. To achieve the latter, C has variables, types, function calls, and structured control flow. Compilers and runtime libraries have some freedom in how they implement these features: they decide where the contents of variables are stored, where the compiler generated data structures are stored, and (to a certain extent) how typed values are encoded. At the same time, C still allows direct access to memory to support efficient system programming. This is done through *pointers*, which are variables containing memory addresses. C allows pointers to be cast to integers – exposing the memory address – and back, and it allows pointers to be manipulated, either by adding an offset or by directly manipulating their byte representation.

The combination of direct access to memory and an undetermined memory layout leads to complications. Below we contrast two snippets of C code to illustrate this. The snippet below is valid C code that changes the second byte of the encoding of an integer: it creates a `char` pointer that contains the address of `foo`, an integer variable. It adds the offset 1 to the pointer, so it now points to the second byte of `foo`, and it changes this byte (on a little-endian architecture `foo` is now `0x01020504`).

```
int foo = 0x01020304;   // We declare an integer
char *p = (char*) &foo; // Pointer p contains the address of foo
p = p + 1;              // Pointer p now contains the next address
*p = 5;                 // We change the value that is stored here
```

The following similar C snippet is not "good" C code. Instead of adding 1 to the pointer, it adds an offset as big as the size of an integer. The pointer therefore no longer points to any of `foo`'s bytes, but it points to the byte just past `foo`.

```
int foo = 0x01020304;   // We declare an integer
char *p = (char*) &foo; // Pointer p contains the address of foo
p = p + sizeof(int);    // Pointer p now contains the address just past foo
*p = 5;                 // We change the value that is stored here
```

What semantics does the last access have? Because the memory layout is not determined by the C language definition, `p` could point to variables other than `foo` (including `p` itself), to compiler generated data, or to compiled code. Writing a value to any of these

can completely change the behaviour of the program. The language definition recognised this by specifying that the code has *undefined behaviour*.

1.2.1 Undefined behaviour

Undefined behaviour is a term used in the ISO C [4, 73] and ISO C++ [72] standards to describe programs that can behave in any way possible. In other words, implementations are not constrained in what behaviour they generate and may even "make demons fly out of your nose" [50].

The rationale behind undefined behaviour is to allow C programs to be efficiently executable on diverse platforms. If the ISO standards had required implementations to raise an exception instead, implementations would have needed to insert runtime checks and keep track of extra information such as the bounds within which a pointer is valid. Undefined behaviour also allows implementations to aggressively optimise code, which we illustrate with the code below. If there exists an execution where `i` is greater than or equal to 10, then the array access in the last line is an out-of-bounds access, giving the entire program undefined behaviour. The compiler can therefore assume that `i` is less than 10, which means the condition in the middle line is always false and the compiler is allowed to omit it. This example also shows that undefined behaviour can have surprising effects, as the program might not print the error message if `i` is out of bounds.

```
int foo[10];
if (i >= 10) printf("Error: i out of bounds");
foo[i] = 0;
```

Below we describe more precisely which memory accesses lead to undefined behaviour. To do this we first consider how pointers relate to *allocations*. Implementations automatically allocate memory for variables, and with the *address-of* operator "&" one can obtain a pointer to such an allocation. Programmers can also allocate memory themselves by calling `malloc` (or its variants), which returns a pointer to that allocation. Although the ISO standard is not explicit about this, the current understanding is that pointers have *provenance*: they stay tied to the same allocation even when their byte representation is manipulated or when offsets are added to the pointer [91, 62]. Out-of-bounds accesses, which we mentioned above, can now be defined as follows. Dereferencing a pointer that no longer points within its corresponding allocation is an out-of-bounds access, leading to undefined behaviour. This is a *spatial* memory error. Dereferencing a pointer while its corresponding allocation is either not initialised or already freed, is a *temporal* memory error, which also leads to undefined behaviour.

There are many other sources of undefined behaviour, such as division by zero, signed integer overflow, and modifying string literals, but these are not relevant for this thesis. We focus on undefined memory accesses because these can often be exploited.

1.2.2 Exploiting undefined memory accesses

Although programs with undefined memory accesses may behave in any way possible according to the ISO standards, there is some regularity in their behaviour in practice. Especially when the undefined access is caused by external input, programs can behave sensibly under normal use but leak information or allow an attacker to execute arbitrary code on certain malicious inputs. This ties into the fundamental problems we described at the start of this chapter: it makes it difficult for mainstream engineering methods to detect the problem, while the security implications are severe. We use the following two (contrived) examples to illustrate these points.

Example 1.1 (Leaking information)**.** The program below contains a secret key (Line 4) and a buffer (Line 5). The main function converts the command line argument to an integer `i` (Line 8), reads the `i`-th value of the buffer (Line 9), and prints the result (Line 10).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int secret_key = 4091;
5  int buffer[ ] = {2, 3, 5, 7};
6
7  int main(int argc, char *argv[]) {
8    int i = atoi(argv[1]); // Convert the command line argument to an int
9    int x = buffer[i];     // Read the i-th element of the buffer
10   printf("%d\n", x);     // Print the read value
11 }
```

If the program is executed with an argument less than 0 or greater than 3, the access at Line 9 is out-of-bounds. The ISO C standard remains vague about what undefined behaviour means in the presence of external input. Instead of a program being entirely defined or entirely undefined, the current understanding is that a program can have defined and undefined *executions*: on inputs that do not cause undefined behaviour the program is defined even though on other inputs the program has undefined behaviour. This means the example program should work as intended on inputs 0, 1, 2, and 3, and print respectively 2, 3, 5, and 7 – the contents of the buffer.

To see how the program behaves on other inputs we observe that implementations typically compile `buffer[i]` to a machine load that loads from the address where the `i`-th element of `buffer` would be if it existed, namely the address of the start of the buffer plus `i` times the memory-width of an integer. On inputs that are outside the bounds of `buffer` this will print whatever happens to be in the memory at that address.

An attacker could exploit this to obtain the secret key. By guessing that the compiler would place the secret key in memory somewhere before the buffer and by trying a few

inputs, we found that running the program with input -4 prints 4091 (using GCC 7.4.0 on Ubuntu 18.04 on an Intel x64 architecture). This means that a simple programming error, which does not affect normal use of the program, is in fact a security bug.

Besides leaking information, out-of-bounds accesses can also be used to change the behaviour of a program, for example by overwriting the values of variables.

Example 1.2 (Changing program logic)**.** The program below asks the user to input a name (Lines 9–10). If the user is authenticated it prints a message (Line 12), but the program does not provide a way to authenticate the user, so this should never happen.

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4  char name[10];
5  bool authenticated;
6
7  int main() {
8    authenticated = false;  // The user is not authenticated
9    printf("Enter name: "); // Ask the user to input a name
10   scanf("%s", name);      // Store the input
11   if (authenticated) {    // Check whether the user is authenticated
12     printf("%s is authenticated\n", name);
13   }
14 }
```

Line 10 contains an out-of-bounds access: if the user inputs more than 10 characters the input does not fit in `name`, and `scanf` overwrites whatever happens to be in the memory after `name`. When we ran this program and input "A…K" (again using GCC 7.4.0 on Ubuntu 18.04 on an Intel x64 architecture) it printed that we were authenticated. This happened because our compiler had put `authenticated` in the memory just after `name`, so `scanf` overwrote `authenticated` with the encoding of "K", which evaluates to true when viewed as a boolean.

Out-of-bounds accesses can also be used to allow an attacker to execute arbitrary code. This works in the same way as the previous example, except that an attacker overwrites compiler generated data instead of program variables. To understand how that works we first consider how the function call at Line 8 in the following program is compiled.

```c
#include <stdio.h>

int triple(int value) {
  return 3 * value;
}

int main() {
  int result = triple(14);        // We call the function "triple"
  printf("Result: %i\n", result); // We print the result
}
```

The compilation of the program is stored in memory as a long list of instructions. When an instruction has been executed the hardware normally executes the next instruction in the list, but *jump* instructions override this behaviour by explicitly giving the address of the instruction that should be executed next. The function call in Line 8 uses a jump instruction to the instructions of `triple`, and `triple` jumps back when it has finished executing. The compiler uses *return addresses* to achieve this: the function call first saves the address of the instruction where the execution needs to continue (in this case the instruction corresponding to Line 9), and then jumps to `triple`. When `triple` has finished executing it jumps to the return address. The return address is saved on the the *stack*, which is a data structure that is also used to store function arguments, local variables, and register values that need to be preserved during the function call.

Some memory safety bugs allow an attacker to overwrite the return address and thus determine which code is executed next. Exactly how this allows an attacker to execute arbitrary code changes over time, as defences against specific attacks are developed. Below we describe this ongoing arms race.

1.2.3 The ongoing arms race

The Morris worm, created in 1988, is one of the first worms that spread through the Internet [149, 148]. It used an out-of-bounds access to write *shell code* to the memory, and it used the same out-of-bounds access to overwrite the return address with the address of this code. This opened a shell on the target machine which the worm used to download and execute a copy of itself. Levy (also known as Aleph One) described in "smashing the stack for fun or profit" the steps involved in such an attack, including how one can determine or guess the address of the injected code [112]. The internet worms we mentioned in Section 1.1 used similar attacks.

As a reaction, many operating systems adopted the W^X policy that forbids memory from being writable and executable at the same time: the attacker can still inject their own code and overwrite the return address, but the architecture will trap when jumping

to the return address. This policy can be efficiently implemented with paged virtual memory, as we described in Section 1.1.

Attackers circumvented the W^X policy by leveraging the code of the program itself. C programs typically use the C standard library *libc*, which contains many functions that are useful in an attack such as the *system* function that executes shell commands. In a *return-to-libc* attack [32], the attacker invokes a library function by writing the function arguments to the stack and overwriting the return address with the address of the function. By overwriting the stack in a particular way an attacker can invoke a series of library functions [168, §3]. These attacks are limited to straight-line code (code without branches), but with *return-oriented programming* one can leverage existing code to do arbitrary computation [79, 137, 127]. Here the attacker analyses the program to find *gadgets*: short sequences of instructions that end with a return instruction. The attacker can combine gadgets by overwriting the return address with a sequence of addresses, since the return address at the end of each gadget ensures that the execution continues with the next gadget. There are efficient algorithms to find a Turing-complete set of gadgets in a target library [69, 38].

One line of defence against return-oriented programming focuses on the return instructions. Using dynamic binary instrumentation one can count the number of instructions between two returns and report an attack if this number is consistently low [26]. More radically, one can compile programs in a way that avoids the return instruction altogether [85]. Unfortunately, return-oriented programming does not fundamentally rely on the return instruction, as indirect jumps can also be used to chain gadgets together, bypassing these defences [22, 16].

Another line of defence tries to prevent an attacker from diverting the control flow. StackGuard [25] places a *canary* between the local variables and the return address. Before a function jumps to the return address it checks whether the canary has changed, and if so the program aborts. A limitation of this approach is that local variables can still be overwritten. If an attacker can overwrite a pointer `p` and a variable `x`, and the program writes `x` to the address where `p` points to (for example with `*p = x`), then the attacker has obtained a *write-anything-anywhere* primitive, which they can use to overwrite the return address without overwriting the canary. This attack can be detected by StackShield [156], as it stores a shadow copy of all return addresses and it checks whether the return address on the stack still matches this copy. However, there are other ways to divert the control flow, for example through the *global offset table* (GOT). The GOT contains the memory addresses of shared library functions, such as `exit`, `free`, and `syslog`. With a write-anything-anywhere primitive the attacker can overwrite the mapped address and determine which code is executed when these functions are called [19, 125].

A third line of defence tries to conceal the address space layout from the attacker, which would mean the attacker does not know where gadgets and library functions are located

in memory. This defence works by randomising the location of the stack, the heap, and shared libraries each time the program starts [152, 173]. Unfortunately, attackers can use information leaks to determine the address space layout during execution [168, 39, 128]. Even fine-grained randomisation that at the start of the program permutes the order of functions, swaps registers, and randomises the location of instructions can be bypassed this way [146].

Three decades after the Morris worm, the arms race is still ongoing. For example, Microsoft estimates that 70% of the vulnerabilities they have patched between 2006 and 2018 have a memory safety issue as their root cause [92], and recently there have been many high-profile attacks exploiting such vulnerabilities [47, 68, 116, 40].

1.2.4 Comprehensive solutions

It seems unlikely that narrowly-tailored defences such as those discussed above will decisively end the arms race. Instead, a comprehensive solution is required. The most comprehensive solution would be to use a memory safe programming language instead of C/C++, or retroactively make C/C++ memory safe, either through software or hardware.

Which memory safe languages are viable alternatives for C/C++ depends on the requirements of the program. If those include interoperability with other programming languages, or fast and/or predictable performance, managed languages such as C# and Java may not be a viable option. For these programs the Rust programming language [88] could be a good compromise: it is an efficient systems programming language that still allows unsafe memory accesses if the programmer marks their code as *unsafe*, but the combination of a type system with ownership and bounds checks on arrays ensures that most code can be written without the *unsafe* keyword. Unfortunately, it would be infeasible to port the billions of lines of existing C/C++ code to another programming language.

Software solutions can retroactively make C/C++ memory safe. CCured [105], for example, adds spatial memory safety using a combination of static analysis and runtime checks: during compilation it determines which pointers cannot stray outside their allocation, and for every other pointer it stores the bounds of its allocation and checks these when dereferencing the pointer. SoftBound [101] uses a similar approach, but avoids memory layout changes by storing the bounds in a disjoint metadata space, instead of in the byte representation of pointers. When combined with SoftBound, CETS [102] also adds temporal safety. It associates a unique identifier with each allocation, the *lock*, and clears it when the allocation is freed; it augments each pointer with a *key* that corresponds to the lock of its allocation; and it checks whether the key still matches the lock when the pointer is dereferenced. SoftBound combined with CETS has a large overhead: 116% on average, ranging up to 300% [102]. Cyclone [74] offers better performance by only

supporting a safe subset of C, which avoids many runtime checks, but legacy programs have to be rewritten to stay within this subset. Checked C [43] also requires source code changes, but offers a smooth adoption path: one can gradually annotate pointers, which are then free of spatial memory violations. For their benchmarks, they modified 17.5% of lines of code on average. Unfortunately, none of these solutions have been adopted in practice, which is likely caused by the need for non-trivial source code changes or their performance overhead.

Hardware solutions that add memory safety to C/C++ promise better performance without requiring any source code changes. For example, HardBound [33] adds spatial memory safety to C/C++ with a performance overhead of 5–10% on average (depending on the variant of HardBound) ranging up to 15–23%. Unfortunately, none of the hardware solutions have been incorporated in mainstream hardware architectures.

There are many other proposed defences against memory safety exploits. Szekeres et al. [151] observe that none of the comprehensive solutions have been adopted in practice and they argue that their performance overhead or lack of compatibility with legacy code is the reason. CHERI, the context of our thesis, aims to change this.

1.3 CHERI

CHERI [170, 164, 161, 157] is a research project that started in 2010 at the University of Cambridge and SRI International. The goal of CHERI is to offer fine-grained memory protection and scalable software compartmentalisation, both with a low overhead and a gradual adoption path, while maintaining compatibility with legacy code. CHERI achieves this by extending commodity hardware architectures with a *capability system*, and by adapting a conventional software stack to make use of this. Before we introduce CHERI's capability system, we first explain capability systems in general.

1.3.1 Capability systems

In the most general terms, a capability is a combination of a *reference* to an object and *access rights* to that object. The access rights of a capability can be exercised by anyone that possesses the capability. Capability systems typically allow operations such as creating less-privileged copies, delegating capabilities to other entities, or deleting them. These ideas go back to 1966, when Dennis and Van Horn described how capabilities can be used to share fine-grained resources between concurrent programs [29]. Their capabilities could refer to memory segments, processes, and input/output devices; and the access rights included ownership and read, write, and execute permissions.

Because anyone that possesses a capability can use its authority, capability systems ensure that capabilities are *unforgeable*, which means they can only be created or modified

through a limited set of operations that adheres to the security policy of the system. Two approaches emerged: a *partitioned* approach and a *tagged* approach [46]. In the partitioned approach capabilities are kept separate from data (here "data" means anything that is not a capability, which includes code). For example, in the Cambridge CAP computer [167] memory segments could either only contain capabilities or only data, and instructions that modify data cannot be performed on capability segments, and vice versa. In the tagged approach each capability-sized and -aligned region of memory has an associated *tag* that specifies whether the region contains a capability or data. Instructions that modify data can be performed on a region that contains a capability, but this will clear the tag, turning the capability into data. The IBM System/38 [67] is an example of this approach. The Cambridge CAP computer used the partitioned approach because the cost of memory at the time was too high to allow for tags. Later hardware systems typically use the tagged approach, because it allows data and capabilities to be stored alongside each other.

Capability systems provide a natural way to follow the principle of least privilege (see Section 1.1.1), because one can efficiently delegate granular permissions. For example, if another program needs read access to a data structure, one can send it a capability to that data structure with just the read permission.

Capability systems also provide a natural way to follow the *principle of intentional use*, which says that a program should explicitly state which authority it uses whenever multiple authorities are available [107]. This avoids the *confused deputy problem* which arises whenever an action is performed on behalf of someone else. The original example [63] is about a paid compilation service that stored billing information in the file *BILL* and allowed customers to provide a file path to receive debugging information. One day a customer provided the file path of *BILL*, and the service overwrote the billing information. In a capability system, the service can use a user-provided capability to write debugging information and its own capability to write billing information, which would ensure it can only write to *BILL* in the latter case.

On the other hand, capability systems do not provide a natural way to revoke permissions. There are three main ways to revoke capabilities, each with their downsides. The first invalidates the object that the capability refers to, but this revokes all capabilities to that object. The second way revokes a specific capability by deleting it together with all its derivations, but finding these capabilities might be difficult. The third creates a layer of indirection for each delegated capability, and invalidates the layer to revoke the delegated capability and its derivations. The downside is that accessing an object needs an additional access to the indirection layer.

1.3.2 Capability systems in practice

Capability systems have had mixed success in practice. Software capability systems have been widely deployed, but their limited support for fine-grained memory protection makes them unsuitable to add memory safety to C/C++. Hardware capability systems, on the other hand, promise fine-grained memory protection with an acceptable overhead, but they have not been incorporated in mainstream architectures.

Implementing a hardware capability system is complex: capabilities interact with other mechanisms, such as virtual memory, memory relocation, processor modes, and hardware exceptions; one needs tool support to be able to experiment with designs; and one has to simultaneously redesign system software to make use of the capability system. Levy provides a detailed comparison of capability systems up to 1984 [84], including academic systems such as the DEC PDP-1 [1], and the Cambridge CAP computer [106, 167]; and commercial systems such as Plessey's System 250 [45], IBM's System/38 [13, 67], and Intel's iAPX 432 [154]. These systems provided valuable insights, but the remaining implementation issues prevented mainstream adoption. Around the same time, Patterson and Sequin's reduced instruction set computer (RISC) showed that moving complexity from hardware architectures to system software reduces design time, reduces design errors, and increases performance [114]. Perhaps because of that, research interest shifted away from hardware capability systems to software systems.

Software capability systems are simpler to implement and easier to deploy because they run on conventional hardware. Examples are the Hydra operating system [172], StarOS [75], EROS [140], and the OKL4 microvisor [64]. OKL4 provides memory isolation and secure communication between guest programs, and has been shipped on more than 1.5 billion devices [44, §2.1]. One of the reasons for this success is the smooth adoption path: one can run legacy software inside a guest operating system, and slowly migrate the most critical parts to separate guest programs.

However, software capability systems have limited support for fine-grained memory protection. Because they run on untagged memory, software systems use the partitioned approach to protect the integrity of capabilities, which means that a data structure that contains both a capability *cap* and some data *d* cannot be naturally represented in memory. This can be solved by adding a layer of indirection: either by storing a data structure that contains *d* and an index *i* to the location of *cap*, or by storing a *directory* capability that contains *cap* and a capability to *d*. Neither of these methods perform well enough to support compiling C/C++ pointers to capabilities, which would be needed to make C/C++ memory safe.

In 1994 Carter et al. reconciled hardware capability systems with the RISC philosophy: their M-Machine [21] implements capability-based memory protection in a load-store architecture with single-cycle instructions. The M-Machine is geared towards software

compartmentalisation and provides a fast context switch, but it has limited compatibility with legacy code: the M-Machine does not support virtual memory based protection, and its capabilities can only refer to regions with a power-of-two size and alignment. The M-Machine has not been widely deployed in practice.

CHERI uses many insights from previous capability systems, while trying to avoid the pitfalls that prevent adoption in practice. Below we describe its capability system.

1.3.3 CHERI's capability system

CHERI uses a hardware capability system that follows the tagged approach. We explain the mechanism of this capability system by describing three interrelated aspects: the definition of capabilities, the operations that capabilities can authorise, and the operations that can be performed on capabilities. In later sections, we explain how the capability system can be used to offer fine-grained memory protection (see Section 1.3.5) and scalable software compartmentalisation (see Section 1.3.6).

Capabilities A capability is a data structure with the fields that are explained below and summarised in Table 1.1 on the next page.

Capabilities have a *Base* and a *Length* field that together specify the memory region that the capability has authority to: a capability can only authorise memory accesses whose footprints are contained in the capability's memory region. The permissions of a capability, which are specified by the *Perms* field, determine what kind of memory accesses the capability can authorise. The possible permissions are summarised in Table 1.2 on page 26. Users can define their own permissions in the *UserPerms* field.

Besides referring to a memory region, a capability also points to a specific address, which is specified by its *Address* field. This address has two purposes. First, it makes capabilities a natural target to compile C/C++ pointers to: the bounds of the corresponding allocation determines the base and length of the capability, while the pointer value itself determines the address. We return to this topic in Section 1.3.5. Second, the address plays a role in capability invocation, which we introduce later in this section.

Capabilities have a *Tag* field that specifies whether the capability is *valid* or not. Only valid capabilities can authorise memory accesses. In our introduction to capability systems we used "data" and "capability" to mean mutually exclusive things, but from now on we use CHERI's terminology where capabilities and data can be interpreted as each other: a capability can be interpreted as data by considering only its byte representation and ignoring its tag, while data can be interpreted as an *invalid* capability whose fields (other than Tag) are decoded from the data. When writing data to memory, the tag that corresponds to the footprint of the access is cleared. In other words, a valid capability becomes invalid if its byte representation is manipulated.

Field	Description
Tag	A bit that specifies whether the capability is valid.
Base	A 64-bit word that specifies the start of the memory region that the capability has authority to.
Length	A 64-bit word that specifies the length of the memory region that the capability has authority to.
Address	A 64-bit word that specifies the address the capability points to.
Perms	A 15-bit word where each bit specifies whether the capability has the corresponding system permission (see Table 1.2 on the next page).
UserPerms	A 16-bit word where each bit specifies whether the capability has the corresponding user-defined permission.
IsSealed	A bit that specifies whether the capability is sealed.
ObjectType	A 24-bit word that specifies the object type. Only sealed capabilities have an object type.
Reserved	An 8-bit word reserved for future use.

Table 1.1: The fields of an uncompressed capability in CHERI

The *IsSealed* and the *ObjectType* fields respectively specify whether a capability is *sealed* and if so, what its *object type* is. Sealed capabilities are temporarily unable to authorise memory accesses. Object types are abstract identifiers that determine how the capability can become unsealed. We describe the sealing mechanism later in this section in more detail.

Finally, capabilities can be *global* or *local*. This is specified by the first bit of the Perms field, although it is not a permission but an information flow label. We explain the flow of capabilities later in this section.

In the original CHERI-MIPS design, capabilities had a total size of 256 bits plus a tag. Since then, the size has been reduced to 128 bits plus a tag, with a compression scheme that exploits the redundancy between Base, Length, and Address, that places some alignment restrictions on capabilities with memory regions that are larger than 256 bytes, and that uses a smaller object type space [171]. In our thesis we use uncompressed capabilities.

Operations authorised by capabilities All memory accesses need to be authorised by a capability. A memory access that loads data can only be authorised by a valid, unsealed capability that has the PermitLoad permission and that contains the footprint of the access in its memory region. The condition for storing data is the same, except

Bit	Name and description
0	IsGlobal is not an actual permission, but an information flow label. It specifies whether the capability is *global* or *local*.
1	PermitExecute: the capability can authorise instruction execution.
2	PermitLoad: the capability can authorise memory accesses that load data.
3	PermitStore: the capability can authorise memory accesses that store data.
4	PermitLoadCapability: if PermitLoad is also set, the capability can authorise memory accesses that load capabilities.
5	PermitStoreCapability: if PermitStore is also set, the capability can authorise memory accesses that store global capabilities.
6	PermitStoreLocalCapability: if PermitStore and PermitStoreCapability are also set, the capability can authorise accesses that store local capabilities.
7	PermitSeal: the capability can authorise sealing capabilities.
8	PermitCCall: the capability can be invoked.
9	PermitUnseal: the capability can authorise unsealing capabilities.
10	PermitAccessSystemRegisters: the capability can authorise system register accesses.
11–15	Reserved for future use.

Table 1.2: The system permissions corresponding to the bits of the Perms field of a capability (see Table 1.1 on the previous page)

that it uses the PermitStore permission; and similarly, the condition for instruction fetch is the same, except it uses the PermitExecute permission.

To be able to load and store capabilities, CHERI introduces memory accesses that preserve tags. These accesses need additional permissions: to load a capability one needs both the PermitLoad and the PermitLoadCapability permissions, to store a *global* capability one needs both the PermitStore and PermitStoreCapability permissions, and to store a *local* capability one needs the PermitStore, PermitStoreCapability, and PermitStoreLocalCapability permissions. Having separate permissions for preserving/not preserving tags, and for local/global capabilities, allows various compartmentalisation scenarios where compartments can share data but cannot share capabilities, or only share global capabilities. We return to this in Section 1.3.6.

Sealing and unsealing capabilities are both operations that themselves need to be authorised by a capability. To seal a capability with an object type t, the capability that is used as authority needs to be valid, unsealed, have the PermitSeal permission, and t must lie in its memory region. Note that the 64-bits address space is here interpreted as the 24-bits object type space by ignoring all non-24-bits addresses. To unseal a capability that has been sealed with an object type t, a similar condition holds, except this uses the PermitUnseal permission.

Finally, accesses to system registers can be authorised by a valid, unsealed capability with the PermitAccessSystemRegisters permission. System registers influence, for example, exception handling and address translation.

Operations on capabilities CHERI defines several instructions that can manipulate capabilities without clearing their tag. We described above how capabilities can be sealed and unsealed, and we mentioned that capabilities can be loaded from and stored to memory. Capabilities can also be copied between registers. The remaining operations are described below.

The Address field of an unsealed capability can be changed to any value within its region of memory, and to some extent to values outside this region. The latter is needed to support a common C/C++ idiom: a pointer that is incremented in a loop can point just outside the bounds of its corresponding allocation when the loop has finished. This does not cause undefined behaviour as long as the pointer is not dereferenced. Likewise, the address of a capability may point outside its memory region as long as it is not used to access memory.

The Base and Length fields of an unsealed capability can be changed, but only in ways that shrink the memory region. Similarly, permissions can be removed from the capability by clearing bits in Perms and UserPerms, but there are no instructions that add permissions. A common idiom in CHERI is to copy a capability and restrict the

memory region and permissions of the copy to create a less-privileged version that can be delegated.

Capabilities can be *invoked*. This takes a pair of capabilities: a capability with the PermitExecute permission, which we call the *code capability*, and a capability without that permission, which we call the *data capability*. These capabilities both need to be sealed with the same object type and have the PermitCCall permission. Furthermore, the address of the code capability must lie within its bounds. When invoking these capabilities, the execution jumps to the address of the code capability and atomically unseals both capabilities. If correctly set up, capability invocation can be used to transition between protection domains: suppose domain A created code and data capabilities *cap* and *cap*$'$ that have authority over A's region of memory, and that are sealed with an object type t that domain B cannot unseal; and suppose that the address of *cap* points to an entry point of A. Domain A can safely pass these capabilities to B: their authority cannot be used by B because they are sealed, and B cannot unseal them. B can only pass them around or invoke them, which would restore the authority of the capabilities, but also return control to A.

1.3.4 CHERI-MIPS

CHERI's capability system *extends* commodity architectures, rather than replacing them. The first prototype version is CHERI-MIPS, which is an extension of 64-bit MIPS [94]. This version has been used to explore the design space and show the feasibility of the CHERI project. There is also a version of CHERI-RISC-V [164, chapter 5], a preliminary sketch of CHERI-x86-64 [164, chapter 6], and Morello, which is a prototype CHERI extension of the Armv8-A architecture [60].

The central design artefact of CHERI-MIPS is its *instruction set architecture* (ISA). An ISA defines the hardware/software interface: the description of the programmer-visible machine state and instruction behaviour. For example, the CHERI-MIPS ISA describes which registers can hold capabilities, how instructions implement the operations we described in the previous subsection, and how the capability system interacts with hardware exceptions, address translation, and debug devices. The CHERI-MIPS ISA is described in prose [164] and in formal specifications [81, 131]. The formal specifications are written in domain specific languages for architecture specifications: one is written in L3 [55] and one in Sail [130]. These specifications are executable, which means that software can run directly above the architecture. Furthermore, they export to theorem prover definitions, giving a mathematically precise definition of the architecture.

Then there is a *microarchitectural* implementation of the CHERI-MIPS ISA. A microarchitecture describes the constituent parts of a processor, for example, its pipeline and cache hierarchies. The CHERI-MIPS implementation is written in Bluespec [110]

and runs on FPGAs. It shows that it is viable to implement CHERI-MIPS with single-cycle instructions, while keeping the overhead moderate: compared to an implementation of MIPS, the implementation of CHERI-MIPS uses 32% more logic elements and its maximum clock speed is 8.1% slower [170, §9].

Finally, there is a software stack above the CHERI-MIPS ISA, including adapted versions of the LLVM compiler [83] and the FreeBSD operating system [90]. The software stack reconciles the capability system with other memory management aspects such as process creation, context switching, page swapping, signal delivery, and static and dynamic linking [27]; and it demonstrates the security benefits of using CHERI's capability system.

The architecture, microarchitecture, and software stack of CHERI-MIPS have been co-designed, which is unusual but proved invaluable for the CHERI project. Every change in the architecture affects the microarchitecture and the software stack in terms of compatibility, security, performance, and viability. By co-designing them, these effects are easier to assess, which makes it feasible to experiment with different architecture designs.

CHERI-MIPS has been used to show that CHERI's capability system can indeed deliver fine-grained memory protection and scalable software compartmentalisation. We discuss these two uses cases in the next two subsections.

1.3.5 Fine-grained memory protection

Capabilities in CHERI can refer to memory regions as small as a single byte, and CHERI's overhead scales gracefully in the number of capabilities in use, making it feasible to protect memory in a granular way. In particular, it makes capabilities a natural target to compile C/C++ pointers to, which adds memory safety to C/C++. We explain the details of this compilation scheme below.

The CHERI C compiler has a *pure-capability* mode in which all pointers are compiled to compressed capabilities [27, 124]. The capability that is provided by the operating system during process creation is subdivided into separate capabilities for the code, data, stack, and heap of the program. Linkers, allocators, and compiler-generated code divide these capabilities further into capabilities for source-level variables, deriving the length of the capability from the C-language type; capabilities for dynamically allocated memory, deriving the length from the size of the requested memory; and capabilities for implied pointers such as stack frames, return addresses, jump destinations, external symbols, and ELF auxiliary arguments. Pointer arithmetic is compiled to instructions that change the address of capabilities, leaving their permissions and memory regions unchanged.

Davis et al. recompiled nearly 800 C programs in the FreeBSD source tree in this pure-capability mode [27, §5.3]. Most programs did not require any source code changes to compile and run successfully. Some changes that were required for the remaining

programs were due to limitations of the pure-capability mode, such as the larger size of capabilities compared to pointers or the new capability registers. Other changes revealed problems in the source code, for example, pointer provenance issues such as casting an `int` to a pointer or creating a pointer to an object from a pointer to an unrelated object, which can cause undefined behaviour and might be exploited.

Compiling programs in pure-capability mode has an overhead: deriving a capability requires more instructions than deriving a virtual address, and compressed capabilities use 64 bits more memory than pointers, leading to more cache misses. On the other hand, the new capability registers relieve pressure from the general purpose registers, which can lead to a slight performance gain. Compared to programs compiled to MIPS, the cycle count increases with a geometric mean of 5% and outliers to 22%, and the number of L2 cache misses increases with a geometric mean of 21% and outliers to 70% on the benchmarks used by Davis et al. [27, §5.2].

Richardson extends the pure-capability mode with *sub-object hardening* [124, chapter 5] to also protect intra-object accesses. Except for a few special cases [124, §5.3–5.4], this provides complete spatial memory protection. Compared to compilation on MIPS, Richardson reports a roughly similar overhead as Davis et al., namely a cycle count increase with a geometric mean of 0.1% and a worst-case of 23%.

Cornucopia [49] adds temporal memory safety to heap allocations when used in conjunction with the pure-capability mode. It places freed memory in quarantine until it has revoked all valid capabilities to that memory region. Cornucopia finds these capabilities by sweeping memory and registers, and revokes them by clearing their tag. To prune the space it needs to search, Cornucopia augments virtual pages with a *full-clean* flag that is set if the page does not contain capabilities, and a *sweep-clean* flag that is set if the page does not contain new capabilities since the last sweep. Then to efficiently find capabilities on a page, Cornucopia only loads the tags of the page, without any data. Furthermore, the revocation pass is split into an initial sweep that is offloaded to another thread, and a final sweep that stops the application threads. The performance overhead of the application threads has a geometric mean of 2.4%, and outliers to 7.9%. Cornucopia traps on all temporal memory safety violations in a benchmark with 1211 tests [49, §VI.C].

1.3.6 Scalable software compartmentalisation

In Section 1.1.1 we introduced software compartmentalisation as a technique to let each part of a program run with only the permissions it needs to function, which restricts the effects that a compromised or malicious part can have on other parts. CHERI's capability system enables compartmentalisation with regard to memory accesses. In other words, it ensures that compartments cannot access memory beyond the memory they were

explicitly granted access to. Furthermore, CHERI provides a way for mutually untrusting compartments to communicate, which we explain later in this section in more detail.

There are two main use cases of software compartmentalisation. The first is to isolate untrusted code such as scripts, macros, and plug-ins, or less-trusted code such as third-party libraries. JavaScript, for example, is untrusted code that is ubiquitous in present day websites. The second use case is to make trusted code more robust against attacks: even security critical code inevitably contains bugs, but compartmentalisation limits the effects of exploited bugs.

There are several differences from the pure-capability compilation mode we described above. First, compartmentalisation only *mitigates* the effects of exploits, while the compilation mode prevents certain memory safety bugs from being exploitable in the first place. On the other hand, compartmentalisation is not restricted to memory safety bugs; it also protects against other vulnerabilities, such as logical errors, and even against malicious code. Furthermore, compartmentalisation can be used to isolate binaries, which avoids the need to trust compilers and allocators, and which is needed in the pure-capability compilation mode.

The mechanism by which CHERI provides memory isolation between compartments is the same as we described before: if a compartment does not posses a capability with authority to a region of memory and permission for the type of access, then the compartment cannot access that region of memory. This holds regardless of the compartment's code, so even if the compartment is compromised or malicious.

Compartments can be set up with *private* memory, which is memory that no other compartment possesses a capability to, and additionally with *shared* memory, which is memory that multiple compartments possess capabilities to. This enables many compartmentalisation scenarios, for example, horizontal compartmentalisation, where compartments share the same code, but work on isolated data; vertical compartmentalisation, where compartments perform different operations on the same instance of data; and assured pipelines, a generalisation of the latter, where a series of compartments performs staged processing of the same data, forbidding communication between compartments that are not adjacent in the series [158, §V]. These scenarios can be fine-tuned by restricting the permissions of capabilities to shared memory: one can control the direction of the exchange by giving write permission to one compartment and read permission to the other; one can prevent the exchange of capabilities by giving permission to access data only; or, similarly, prevent the exchange of local capabilities by only giving permission to store global capabilities. If capabilities to the stack of a C/C++ program are always local, the latter can be used to ensure that capabilities to the heap can be exchanged, but capabilities to the stack cannot.

Besides sharing memory, compartments can also share control over an execution. We call a transition of control a *protection domain switch*. There are two ways to switch

protection domains in CHERI, both of which cause a non-monotonic change in the available privileges. The first way is through a trusted intermediary. A calling compartment transitions into the intermediary by raising a specific hardware exception, which, like all hardware exceptions, jumps to an exception handler and makes the *kernel code* and *kernel data* capabilities available for use. The intermediary can use these capabilities to access memory that the compartments do not have access to, for example to maintain a protected stack and/or to load the capability of the called compartment. When the intermediary is finished, it makes the kernel capabilities unavailable and jumps to the compartment that is called. The second way is through capability invocation, which we introduced at the end of Section 1.3.3: by giving compartment A a pair of sealed capabilities that have authority to compartment B's memory, that point to an entry point of compartment B, and that are sealed with an object type that A cannot unseal, we allow A to transfer control to B while ensuring that A cannot access B's private memory. After the transition, B can access its private memory because the capabilities are unsealed during invocation. Here, A has to ensure it does not leak capabilities to its own private memory through registers or shared memory. Both ways of sharing control are non-hierarchical, allowing A and B to transfer control back and forth, despite being mutually untrusting.

Compartmentalisation in CHERI scales gracefully in the number of compartments, the degree of memory sharing, and the rate of protection domain switches. The reason behind the first two points is the scalability of the capability system, and the reason behind the last point is the modest overhead of a protection domain switch: calling a compartment and returning back, while clearing and restoring registers that should not be leaked, costs around 580 cycles for the method that uses a trusted intermediary, which is modest compared to the cost of >17,000 cycles for domain switches between compartments that are isolated via OS processes [160, §7]. The protection domain switch that uses capability invocation has an even lower overhead, as it avoids the intermediary.

1.3.7 Will CHERI become mainstream?

Integrating a capability system in mainstream architectures requires a considerable investment, but we have a strong incentive to do so: our inability to create secure software is related to the limitations of our current hardware architectures. Furthermore, advances in hardware engineering such as FPGAs, executable architecture specifications, and synthesisable hardware description languages make it feasible to design ambitious hardware, and the lower cost of hardware makes the chip-size and memory overheads acceptable.

However, we should acknowledge that this does not guarantee success. In 1976 Denning used the same arguments to predict the rise of hardware capability systems [28]: "Now, in the middle 1970s, we have come to appreciate the serious limitations of our 1960s machines; we are much more sensitive to the issues of security and reliability; we

are receptive to proposals for more secure and reliable systems; and, most importantly, we have at hand the technology to construct what once was considered ambitious and expensive hardware. The outlook is optimistic." Yet, 45 years later, hardware capability systems are not widely deployed.

Because of this historic legacy, CHERI has a stronger focus on adoption in practice than previous hardware capability systems, which is embodied in CHERI's incremental adoption path. The start of the adoption path is that CHERI is fully backwards compatible. Legacy memory accesses are implicitly authorised by a capability in a fixed register, namely the *default data capability* (DDC), and virtual memory based memory protection is still supported. This allows MIPS binaries to run unmodified on CHERI-MIPS. Then, with moderate effort, one can make vulnerable systems more robust: one can isolate legacy binaries with wrappers that restrict the DDC, one can prevent spatial memory vulnerabilities by recompiling C/C++ programs in the pure-capability mode [27, 124], and one can add temporal safety for heap allocations by enhancing allocators with Cornucopia [49]. Finally, with more effort, one can increase performance by replacing existing virtual memory based protection, for example to let guest programs in a hypervisor run in the same address space, and one can increase security and reliability of trusted code such as operating system kernels, interpreters, JIT compilers, and cryptographic libraries by compartmentalising them.

Although it is too early to tell whether CHERI will become mainstream, its adoption in practice seems promising so far. Most notably, Arm and the UK government are involved in the Morello program, which is developing the eponymous prototype CHERI extension of the Armv8-A architecture, along with a processor implementation, development board, and software. Morello is part of the £187 million Digital Security by Design Challenge, with £70 million from the Industrial Strategy Challenge Fund [30], £50 million from Arm [60], and the rest from other industrial contributions [31]. Arm and Cambridge agreed to make capability essential IP available for use without restriction [166], enabling other industrial vendors to also develop a CHERI extension.

In the rest of this thesis, we focus on one of the aspects that will determine CHERI's success in practice, namely its correctness.

1.4 The correctness of CHERI

Since CHERI's goal is to protect security critical software, its own correctness is crucial. Ensuring CHERI's correctness is difficult for reasons that are similar to the fundamental problems we described at the beginning of the introduction: a small bug in the capability system can nullify all the protection that CHERI offers, and mainstream engineering methods are not suited to find small bugs in corner cases. CHERI's engineering methods

go beyond mainstream methods, which we describe below, allowing a small team to create the CHERI-MIPS architecture, implementation, and software stack. Yet, these methods are still insufficient for a system as critical as CHERI, and indeed could not prevent some serious vulnerabilities in CHERI-MIPS. We discuss the strengths and limitations of CHERI's engineering process below, and start with CHERI's use of formal architecture specifications.

1.4.1 Formal architecture specifications

Traditionally, architectures are developed using prose and pseudocode specifications, and implementations are verified with hand-written test-suites. In the CHERI engineering process this is replaced with formal specifications and auto-generated tests. These formal specifications are written in the domain specific languages L3 [55] and Sail [130], as mentioned in Section 1.3.4. The specification of CHERI-MIPS [81] was originally written in L3 and has now been ported to Sail [131], and the specification of CHERI-RISC-V [132] is written in Sail. Morello is developed in Arm's internal specification language ASL, which is similar to L3 and Sail, and which can be translated to Sail in a largely automated way [6].

Formal architecture specifications have several benefits over prose specifications. The first benefit is that they improve the quality of the specification. L3 and Sail are parsed and type-checked, catching errors that are easy to make in prose. Furthermore, they allow no ambiguity, which avoids misunderstandings between designers, implementers, and users of the architectures.

The second benefit is that formal architecture specifications can serve as oracles for hardware testing. L3 and Sail automatically generate emulators, variously in SML, OCaml, and C, which allows programs to be executed directly on the specification. Hardware implementations can be tested by running them alongside the specification and comparing their traces, which avoids the need to interpret a prose specification and manually curate test outcomes. This, in turn, makes it possible to auto-generate tests. With a combination of symbolic execution of the formal specification and constraint solving, it is possible to generate pseudo-random sequences of instructions that achieve good coverage [20].

A similar benefit holds for testing the software stack. When software is executed on a hardware implementation and something goes wrong, it may not be clear whether the software makes an invalid assumption or the implementation fails to meet the specification. Executing the software directly on the formal architecture specification helps answering this question. Furthermore, one can run the software stack as soon as the specification changes, without having to wait for a new hardware implementation. The emulators generated by L3 and Sail perform well enough to allow executing a software stack. For

example, the CHERI-MIPS emulators run at 300–400 KIPS, which is fast enough to boot FreeBSD in around four minutes.

These benefits would be defeated if the formal specifications diverged from the intended architectures. To prevent this, the specifications of CHERI-MIPS and CHERI-RISC-V are written by the same engineers and researchers that would otherwise write the prose specification. Furthermore, the formal specifications of instructions are included verbatim in the documentation, replacing pseudocode. This is possible because L3 and Sail are designed to read like traditional prose specifications. In particular, they do not require a formal background to write or understand. Finally, the fact that the formal specifications are executable makes it easier to discover unintended behaviour. The formal specification of Morello is not directly developed in L3 or Sail, but here divergence is avoided by directly translating Arm's authoritative specification, as mentioned above.

Compared to traditional engineering methods, CHERI's engineering methods give more confidence that hardware implementations and software stacks conform to the corresponding architecture. This leads to the next question, namely whether CHERI architectures indeed support the security use cases of CHERI.

1.4.2 Reasoning about security use cases

Reasoning about security use cases typically involves universal quantification over code. For example, reasoning about compromised or malicious compartments, the pure-capability compilation mode, or the sweeping revocation algorithm Cornucopia all involve arbitrary code. In principle, architectures provide enough information to reason about these use cases, but this may not be feasible in practice: architectures are large artefacts containing hundreds of instructions, described in thousands of lines of formal specification, and every instruction needs to be considered. At this abstraction level, manual reasoning is infeasible, and automated reasoning, such as symbolic execution or model checking, quickly suffers from a combinatorial explosion.

The designers of CHERI use their intuitive understanding of CHERI when reasoning about arbitrary code, and some of their intuitions have been captured in *security properties*. These properties describe CHERI's protection mechanism at a high level, abstracting away from the behaviour of individual instructions. The designers ensure that CHERI architectures satisfy these security properties, and they take care not to inadvertently break them when adding new features. The security properties are described in high level prose in the documentation of CHERI [164].

Unfortunately, CHERI's security properties are less precise than they could be, and omit crucial details. To a large extent, these problems are inherent to prose security properties.

1.4.3 The problems with prose security properties

The first problem with prose security properties is that they are prone to ambiguities, which may lead to security vulnerabilities if users, designers, and implementers misunderstand each other. To illustrate this problem, we identify ambiguities in the prose definition of a fundamental property of CHERI's capability system, namely *capability monotonicity*. The documentation defines this as the property that "new capabilities must be derived from existing capabilities only via valid manipulations that may narrow (but never broaden) rights ascribed to the original capability" [164, §2.3.4]. But what constitutes broadening the rights of a capability? Broadening its bounds and increasing its permissions are given as examples, but does unsealing a capability also broaden its rights? This is left unclear. The documentation continues with: "monotonicity allows reasoning about the set of reachable rights for executing code, as they are limited to the rights in any capability registers, and inductively, the set of any rights reachable from those capabilities". This describes an upper bound of the rights that (untrusted) code can use if we allow it to execute arbitrary instructions. This upper bound is defined as the rights that are transitively reachable from the capabilities in the capability registers. However, the documentation does not define when a right is reachable from a capability, so one cannot know exactly what this upper bound is. This is further complicated by the following statement: "the two notable exceptions to capability monotonicity are invocation of sealed capabilities and exception delivery", and it continues with: "where non-monotonicity is present, control is transferred to code trusted to utilize a gain in rights appropriately". Is this code "trusted" in the sense that we have to trust it? Can untrustworthy code perform non-monotonic derivations to gain rights? It is difficult to see why non-monotonicity does not defeat CHERI's protection mechanism.

The second problem with prose security properties is that it is difficult to establish whether they actually hold. Prose properties cannot be experimentally validated, for example by testing or model checking, and they are not susceptible to mathematical proof. To illustrate this problem we discuss a serious security bug in the previous version the CHERI-MIPS ISA [162, chapter 5]. The CLC instruction only loads a capability *cap* if the capability *auth* that is used as authority has the *PermitLoadCapability* permission. This behaviour was changed in the mentioned version: if *auth* does not have that permission, the instruction still loads the byte representation of *cap*, but not its tag. This seems harmless, because by stripping the tag the loaded capability becomes invalid. However, capabilities that neither have the *PermitLoad* nor the *PermitLoadCapability* permission can now be used to load data via this instruction, bypassing CHERI's protection mechanism. A regression test that checks whether the architecture still satisfies the security properties would conceivably have spotted this bug, but this is not possible with prose security properties.

The problems with CHERI's security properties are a threat to CHERI's success in practice. Precisely which security guarantees CHERI offers remains unclear, and it is an open question whether CHERI architectures such as CHERI-MIPS, CHERI-RISC-V, and Morello indeed provide these guarantees. It is therefore conceivable that they contain serious security vulnerabilities, such as the one shown above, which would then be inherited by all conforming hardware implementations. The impact of vulnerabilities is aggravated by the fact that they might not be fixable without replacing the hardware, which the Pentium bug [117] showed to be costly.

1.5 Thesis

Our thesis is the following:

> Formal statement and mechanised proof of security properties can be made feasible for a production-scale capability-enhanced instruction-set architecture (CHERI-MIPS), increasing confidence in its correctness and security.

1.5.1 Defining formal security properties

It is possible to define formal security properties about CHERI architectures because their formal specifications export to *interactive proof assistants*: variously Isabelle/HOL [111], HOL4 [59], and/or Coq [14]. Interactive proof assistants allow us to state new definitions based on the exported architecture definition, and they enable mechanised proofs about these definitions, which we explain later in more detail. We define security properties for CHERI-MIPS, the longest-developed version of CHERI. These properties are based on the L3 specification of CHERI-MIPS instead of the Sail specification for reasons we describe in Section 1.5.3. We use Isabelle as the proof assistant.

Our first set of security properties forms a new abstraction layer of CHERI-MIPS (see Chapter 3). This abstraction explains execution steps in terms of nine abstract actions: one for each type of memory access (loading/storing with/without tags), one for each type of capability manipulation (restricting, sealing, unsealing, invoking), and one for hardware exceptions. For each action, we define a property that states under what conditions the action can be performed, and what effects it has. We connect CHERI-MIPS to this abstraction by mapping its instructions to abstract actions. Through this mapping, the properties about abstract actions become security properties about CHERI-MIPS.

As part of this abstraction, we define an order over capabilities, capturing when the authority of one capability is contained in the authority of another capability (see Section 3.2). This order clarifies what "broadening the rights of a capability" should mean in the prose definition of capability monotonicity. The abstract actions precisely state

when this non-monotonicity can occur, clarifying the prose statement that "where non-monotonicity is present, control is transferred to code trusted to utilize a gain in rights appropriately". The abstraction also states properties that have no prose counterparts in the CHERI documentation, but that are nonetheless crucial to the capability system.

We use the abstraction to reason about a security use case of CHERI (see Chapter 4). We first characterise which capabilities a (potentially compromised or malicious) compartment could access or construct if it is allowed to execute arbitrary code, and we state related properties about which part of the memory and which registers the compartment can overwrite. This captures what "reachable rights for executing code" should mean. We then turn to a simple compartmentalisation scenario where a compartment is isolated from the rest of the program. Specifically, we consider the part of the execution from the moment the compartment starts executing until it ceases the execution to another compartment. We define the guarantees that CHERI-MIPS offers in this scenario, and under which assumptions they hold.

When defining formal security properties one should consider which mathematical concepts to use to express them: more sophisticated mathematics can let one state properties closer to one's intention, or more elegantly, but it can also make them less accessible. For example, we cannot assume that the intended audience for security properties of CHERI-MIPS is familiar with higher-order functions or computation tree logic (CTL). In the case of higher-order functions we use them nevertheless, to let us define our abstraction independently from CHERI-MIPS's semantics, but we avoid CTL in favour of spelling out the properties in terms of concrete traces, because we thought that accessibility was more important here than stating the properties succinctly.

Having defined formal security properties, we discuss below how we verify that the CHERI-MIPS architecture indeed satisfies them.

1.5.2 Proving formal security properties

The first fundamental problem we described at the beginning of the introduction stated that mainstream engineering methods are not suited to find small bugs in corner cases. This is problematic because attackers can deliberately guide a system to these corner cases, triggering the bug whenever they want. Mathematical proofs can solve this problem, because they consider every possible corner case, not just those exercised by the tests. We therefore use a mathematical proof to verify that the CHERI-MIPS architecture satisfies our security properties (see Chapter 5).

There are three challenges in proving security properties for production-scale architectures. The first is that architectures contain many low-level details that are easy to miss, so checking a paper proof by hand would be unusably error-prone. To solve this problem we *mechanised* all our proofs in Isabelle/HOL. A mechanised proof is expressed in a formal

proof language, typically combining manual and automated reasoning steps. Isabelle, like other interactive proof assistants, defines such a formal proof language, and automatically verifies whether proofs written in that language indeed form a valid mathematical proof. To minimise the trusted computing base, Isabelle has an LCF-style inference kernel [58]: all reasoning steps generate series of simple logical inferences that are checked by a small kernel. This includes automated reasoning steps, which means that the tactics they use do not need to be trusted.

The second challenge is the scale of the proof development. Production-scale architecture specifications are large, and any part of the architecture could potentially break security properties, even if it does not directly interact with the protection mechanisms. For example, in CHERI-MIPS, the majority of the 200-odd instructions do not interact with capabilities, but could still break our security properties. To solve this challenge we developed automated proof tactics, tailored to L3 specifications, that reduce the need for manual reasoning steps. We used Eisbach [87], an extension of Isabelle's proof language, for these. Our custom tactics can automatically prove the security properties for most CHERI-MIPS instructions that do not directly interact with the capability mechanisms, and significantly simplify the proofs for the others. It is worth repeating that these custom tactics do not need to be trusted, as their output is verified by Isabelle's kernel.

The third challenge is that architectures keep evolving. As a research architecture, CHERI has evolved rapidly, but industrial architectures such as Intel 64/IA-32 and Armv8-A do too, with new versions approximately every six months. It would be infeasible to continuously re-check whether a manual proof still holds for updated versions of the architecture, but automated theorem provers can do this automatically, and will point out any places where the proof fails. Our custom proof tactics are reasonably resilient to changes in the specification. To further reduce the effort needed to change our proofs, we use Python scripts to generate the statements and proofs of many lemmas. Again, it is worth pointing out that these Python scripts do not need to be trusted, as Isabelle verifies the generated proofs and checks whether the generated lemmas are used correctly in the complete proof.

1.5.3 Scope and limitations

The level of confidence that mechanised proofs give is often simultaneously over- and underestimated. A common concern is whether mechanised proofs are indeed logically valid. As mentioned above, proofs that are mechanised in Isabelle are verified by Isabelle's kernel. This kernel is based on more than 40 years of research in automated theorem proving, counted from Gordon et al.'s paper about LCF [58], and has been used for high-profile verification projects. This achieves "a degree of trustworthiness of formal,

machine-checked proof that far surpasses the confidence levels we rely on in engineering or mathematics for our daily survival", as Klein observed [77].

However, a proof only shows the validity of the proven statement, and nothing else. In particular, a theorem about a specification of an object does not necessarily hold for the object itself: the specification may leave out aspects that could be relevant for the result. For example, architecture specifications usually abstract away from timing behaviour and power consumption, so theorems about the architecture cannot talk about possible side-channel information flow via these. To avoid overestimating the assurance that a theorem gives, one should carefully read the statement and be aware of the specification that it is proven in. In our case, the mechanised proofs show that the security properties hold in the Isabelle/HOL export of the L3 specification of the architecture of CHERI-MIPS. Below we discuss how this result relates to the L3 specification itself, to the intended architecture of CHERI-MIPS, to hardware implementations of CHERI-MIPS, and to the CHERI architecture in general, but establishing these relations is not in the scope of our research.

The CHERI-MIPS architecture has two specifications, one in L3 and one in Sail, as mentioned in Section 1.4.1. Our properties are defined in terms of the L3 specification simply because the Sail specification did not exist when we started our work. Other architecture specifications have been developed in Sail because it has a more sophisticated type system and contains features that ease a translation from ASL, Arm's internal specification language. To keep CHERI-MIPS in line with the other versions of CHERI, it has been ported to Sail. L3 and Sail are similar enough that our discussion below also holds for the Sail specification.

The L3 specification is no longer maintained since we finished our proofs. The last version is from October 2019 and corresponds to version 7 of the CHERI-MIPS ISA [164], except for the following instructions that were introduced later: CSetAddr, which changes the address of a capability; CCheckTag, which raises an exception if the tag of a capability is unset; CGetCID and CSetCID which respectively store or load the architectural compartment ID; and CLCBI, which loads a capability and is a close variant of CLC, which is included in the L3 specification. The L3 specification includes the experimental instructions CBuildCap and CCopyType, but no other experimental instructions.

We trust that the translation from L3 to Isabelle/HOL is correct. The translation to HOL4 has been extensively used in other verification projects, such as CakeML [54] and seL4 [136]. This gives some confidence in the translation to Isabelle/HOL, since their languages are similar. To improve confidence in the translation, one could use Isabelle's code generation features to run tests on the L3 specification and the Isabelle/HOL export and compare their traces, or one could visually compare the L3 specification to the Isabelle/HOL export.

CHERI's engineering methods ensure that the L3 specification has a close resemblance to the intended architecture of CHERI-MIPS, as discussed in Section 1.4.1. Nevertheless, there are some subtle differences. For example, the TLBWR instruction architecturally writes a random TLB entry, but the CHERI-MIPS hardware implementation writes a certain entry based on a counter. To facilitate comparing traces of the hardware implementation and the L3 specification, the L3 specification deviates from the architecture and writes the entry based on the same counter. As far as we know, these differences do not affect our security properties.

The CHERI-MIPS architecture specifies that certain instructions have *unpredictable behaviour* when used improperly. For example, 32-bit arithmetic instructions have unpredictable behaviour when used with 64-bit operands. Unpredictable behaviour means that implementations may non-deterministically change the machine state and produce garbage values, but they may not "modify memory or capability registers in a way that allows the capability mechanism to be bypassed" [164, §9.1]. The L3 specification of CHERI-MIPS defines when an instruction has unpredictable behaviour, but it does not formalise the semantics of that behaviour. Because this semantics plays a role in our proofs, we formalised unpredictable behaviour ourselves. Since it is not clear what "bypassing the capability mechanism" in the prose specification precisely means, we formalised unpredictable behaviour by requiring that it does not modify memory or capability registers at all, and added that it should not modify certain system registers.

As mentioned before, the CHERI-MIPS architecture abstracts away from timing behaviour and power consumption, so our properties cannot talk about possible side-channel information flow via these. Furthermore, the architecture is a loose specification, allowing variation in the architectural visible behaviour of implementations, as shown by the TLBWR instruction and unpredictable behaviour. Therefore our properties cannot talk about architecturally visible information flow: a (compromised or adversarial) hardware implementation could leak information while conforming to the architecture by exploiting this looseness. Our properties do exclude architecturally visible *capability* flow, which is necessary, even if not sufficient, to prevent information leaks. There is ongoing non-formal work exploring side-channels in CHERI [163].

The L3 specification of CHERI-MIPS only covers the uniprocessor case. To cover the multiprocessor case, the specification would need to describe which parts of an instruction happen atomically, allowing intra-instruction concurrency between other parts, and it should be combined with a realistic relaxed memory model. The challenges of defining and proving security properties in the multiprocessor case are largely orthogonal to the challenges that we solve with our research.

We use the version of the L3 specification that uses uncompressed capabilities. Assuming that the compression scheme is correct, our properties should also hold for the version that uses compressed capabilities.

The phrasing of our security properties is specific to CHERI-MIPS. For example, it refers to capability registers that are only present in CHERI-MIPS. However, we expect the security properties for other CHERI architectures to be conceptually the same.

1.6 Collaborations

This thesis builds upon the work of many others, and in particular on the CHERI-MIPS architecture developed by the CHERI team [170, 164, 161, 157], its L3 specification developed by Alexandre Joannou, Anthony Fox, Michael Roe, Matthew Naylor, and Brian Campbell [109, §III][81], and its Isabelle/HOL export developed by Anthony Fox [55]. Our formal security properties are inspired by many discussions with the CHERI team, especially with Robert N. M. Watson, Simon W. Moore, Alexandre Joannou, Michael Roe, Jonathan Woodruff, and Peter Sewell. The formal statement of our security properties, their factorisation through an abstraction, and their proofs are our own work.

1.7 Publications

The following peer-reviewed paper directly contributed to this thesis.

- Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *S&P 2020: IEEE Symposium on Security and Privacy*, pages 1003–1020, 2020. https://doi.org/10.1109/SP40000.2020.00055.

An early version of the paper above has been published as Technical Report UCAM-CL-TR-940, https:/doi.org/10.48456/tr-940.

During my PhD, I also worked on C/C++11 concurrency, with the following peer-reviewed papers. Those works are not reported in this thesis.

- Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. In *OOPSLA 2016: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 111–128, 2016. https://doi.org/10.1145/2983990.2983997.
- Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *POPL 2017: Proceedings of the 44th ACM*

SIGPLAN Symposium on Principles of Programming Languages, pages 429–442, 2017. https://doi.org/10.1145/3009837.3009839.

- Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *PLDI 2016: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–15, 2016. https://doi.org/10.1145/2908080.2908081.
- Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *ESOP 2015: 24th European Symposium on Programming*, pages 283–307, 2015. https://doi.org/10.1007/978-3-662-46669-8_12.

Chapter 2

The CHERI-MIPS architecture

In this chapter we describe the context of our thesis: the base MIPS architecture (see Section 2.1), the extension to the CHERI-MIPS architecture (see Section 2.2) developed by the CHERI team [170, 164, 161, 157], its L3 specification (see Section 2.3) developed by Joannou et al. [109, §III][81], and its Isabelle/HOL export (see Section 2.4) developed by Fox [55]. In Section 2.5, which is our own work, we augment the specification by formalising aspects that were only defined in prose or left implicit in CHERI's documentation, but which are necessary to state and prove our security properties.

2.1 The MIPS architecture

MIPS64 [93, 94, 95] is a conventional 64-bits RISC architecture. Its documentation totals 1066 pages, so it is not feasible to describe MIPS in full detail here. Instead, we give a brief overview of the parts that are most relevant to our thesis. It is good to note, however, that the parts of MIPS we do not explain are still relevant to our proofs: the security properties we define in this thesis refer to arbitrary executions, so any behaviour that MIPS can exhibit, as far as it is specified in L3, could potentially break these properties.

The MIPS architecture, like most architectures, has a non-deterministic semantics. Instructions behave deterministically when they are used as intended, but otherwise they can cause *unpredictable* behaviour. The effects of unpredictable behaviour can differ between implementations, but also vary over time on the same implementation. Unpredictable operations may cause arbitrary exceptions, and may read, write, and modify any memory or internal state that is accessible from the current processor mode [94, §1.2.1]. Software is not supposed to cause unpredictable behaviour, but compromised or malicious software may intentionally do so.

Below we give an overview of the MIPS instructions that are used in CHERI-MIPS, and we highlight some differences between MIPS and other architectures.

2.1.1 Arithmetic

MIPS defines around 70 arithmetic instructions. Following the RISC paradigm, these instructions do not access memory, but only operate on registers. They typically have a 32- and a 64-bit variant; the 32-bit variants are included for compatibility and cause unpredictable behaviour when used on 64-bit operands. Most arithmetic instructions also have a signed and an unsigned variant: unsigned integers wrap on overflow, but signed integers cause a hardware exception on overflow. The operations that are supported include addition, subtraction, multiplication, integer division, bit shifts, comparisons, and logical operations such AND, OR, XOR, and NOR. Floating-point instructions are optional in MIPS and are not formalised in the L3 specification of CHERI-MIPS.

2.1.2 Memory accesses

MIPS defines variants of load and store instructions for 8/16/32/64-bit words, for aligned/unaligned words, and for words interpreted as signed/unsigned integers. Unaligned words may span two non-contiguous pages and need to be accessed in pieces. MIPS also defines *linked loads* and *conditional stores* which can be used to synchronise multi-threaded programs.

Unlike other RISC architectures, MIPS uses a software managed TLB. On a TLB miss, the architecture raises a hardware exception and stores the address that could not be translated in an exception register. System software should handle this exception, insert the correct translation in the TLB, and restart the aborted instruction. To allow this, MIPS includes instructions that read and modify the TLB. These are only available in kernel mode, which is the most privileged processor mode.

The semantics of a memory access depends on many details. Below we summarise the most important cases.

- The virtual address space is divided into nine segments [95, §4.3]. Some segments are only accessible in kernel mode, some also in supervisor mode, while others are accessible in all processor modes.
- Virtual addresses can be *mapped* or *unmapped*. Unmapped virtual addresses are translated by discarding the most significant bits, while mapped addresses are translated using the TLB. Whether an address is unmapped depends on the processor mode, the segment it lies in, and the current exception level. It is possible that the same virtual address is mapped in user or supervisor mode, while unmapped in kernel mode.
- Normally, TLB entries are associated with an address space identifier, ensuring memory isolation between processes, as discussed in Section 1.1, but this behaviour can be overridden.

- If the TLB is not managed correctly, multiple entries could match a virtual address, causing unpredictable behaviour.
- TLB entries may specify that only certain types of accesses are allowed. This mechanism can be used to implement the W^X policy that prevents code-injection attacks, as explained in Section 1.2.3.
- Physical addresses may be in use by a JTAG UART component or a programmable interrupt controller (PIC). JTAG UART components allow on-chip instrumentation, typically used for debugging, and PICs detect, prioritise, and forward interrupts from hardware devices to the processor.
- Certain combinations of cacheability, coherency, and access types cause unpredictable behaviour, for example, the combination of linked loads and uncached addresses.

2.1.3 Branches

In MIPS, the effect of a branch is delayed by one instruction: after the branch, the architecture executes the next instruction in memory as if the branch did not happen. This instruction is in the *branch delay slot*. Not all architectures have a delay slot, for example, Arm and RISC-V do not. Executing another branch in the delay slot causes unpredictable behaviour. When the instruction in the delay slot has finished executing, the branch takes effect and the program counter (PC) is changed to the destination of the branch.

As usual, MIPS defines branches that just change the PC, and branches that also save the address of the instruction after the delay slot, which can be used to jump back. This address corresponds to the return address we discussed in Sections 1.2.2–1.2.3.

2.1.4 Hardware exceptions

MIPS uses hardware exceptions for several mechanisms, for example, managing the TLB, responding to interrupts, and stopping at debugging breakpoints. To enable restarting the original sequence of instructions, exceptions store the PC to the *exception PC* (EPC). If the instruction at the time of the exception is in a delay slot, the PC of the branch is saved instead. The ERET instruction can be used to return from an exception: it clears the exception level (EXL) bit, reverts the processor mode, and restores the PC from the EPC.

2.1.5 Configuration

A MIPS machine can be configured in great detail. For example, the TLBWR instruction that overwrites an arbitrary TLB entry does not overwrite *wired* TLB entries, and the

ratio between wired and non-wired entries can be dynamically adjusted. One configures the machine by copying the relevant system register to a general purpose register (GPR), changing the setting with ordinary arithmetic instructions, and then copying the modified setting back to the system register. Instructions that copy between GPRs and system registers can only be executed in kernel mode.

2.2 CHERI's extension to MIPS

Extending a commodity architecture with CHERI's capability system is a delicate design process. First of all, the extension has a broad impact on the architecture that goes beyond changing its memory access mechanism. Furthermore, architectures have their own design philosophy, and both the changes to the architecture and any new instructions should follow this philosophy as much as possible.

Below we describe how CHERI extends MIPS [164, 157]. We first describe the new architectural state that enables capabilities to be stored in memory and in registers. We then describe how CHERI changes the semantics of MIPS instructions, which includes changes to MIPS's exception mechanism, the requirements of its privileged instructions, and its virtual memory based protection. Finally, we introduce the new CHERI instructions that, among other things, implement the operations we described in Section 1.3.3 in a MIPS-compatible way.

2.2.1 Tagged memory and capability registers

In Section 1.3 we discussed different approaches to store capabilities in memory, and we mentioned that CHERI uses the tagged approach. This means that each capability-sized and -aligned region of memory is extended with a tag that distinguishes between valid and invalid capabilities. Recall that data and *invalid* capabilities are interchangeable: interpreting data as a capability yields an invalid capability, and interpreting a (valid or invalid) capability as data yields its byte representation, but not its tag. The size of a capability in memory depends on the version of CHERI-MIPS: 256 bits for uncompressed, and 128 bits for compressed capabilities.

CHERI-MIPS's capability registers adhere to the following design decisions. CHERI-MIPS does not compress capabilities in registers, even in versions that compress capabilities in memory. The downside is that registers need to be 256-bit instead of 128-bit, but the benefit is that instructions can use the authority of capabilities without needing to uncompress them first, which is difficult to achieve in single-cycle instructions. Furthermore, CHERI-MIPS adds new 256-bit registers instead of extending existing 64-bit registers to 256-bit. This removes uniformity between registers, but avoids a 256-bit wide path through the main pipeline in the hardware implementation. Finally, CHERI-MIPS

uses tagged 256-bit registers, so they can hold both valid capabilities and data. This makes it possible to copy between memory and registers in a tag-agnostic way.

These design decisions lead to tagged, 256-bit versions of existing MIPS registers: there are general purpose capability registers (GPCRs), a program counter capability (PCC), and an exception program counter capability (EPCC). CHERI also adds new capability registers that do not have an equivalent in MIPS. For example, it adds the default data capability (DDC), the kernel code capability (KCC), and the kernel data capability (KDC). We explain the purpose of these below.

2.2.2 Semantic changes to MIPS

CHERI requires that each memory access is authorised by a capability, so the semantics of memory accesses in MIPS needs to change. At the same time, CHERI-MIPS should be able to run unmodified MIPS binaries. These two design goals are reconciled by implicitly using the authority of the capability in a fixed register: instruction fetch is authorised by the program counter capability (PCC) and MIPS load and store instructions are authorised by the default data capability (DDC). If these capabilities have all permissions and have authority over the entire virtual address space, legacy MIPS binaries behave the same on CHERI-MIPS as on MIPS. Alternatively, one could restrict the PCC and DDC to the memory region that the legacy binary needs access to. If a malicious or compromised binary then tries to access any other memory, the hardware will trap.

CHERI changes MIPS's ring based protection of privileged instructions. Privileged instructions can be used to circumvent the capability system, for example by changing virtual address space mappings. In MIPS, these instructions can be executed in kernel mode, but to avoid having to trust all kernel code, CHERI-MIPS introduces a new requirement: the PCC needs to have the SystemRegisterAccess permission. The kernel can then be compartmentalised in parts that need that permission, which we consequently still need to trust, and parts that do not need that permission. For backwards compatibility, kernel mode is still a requirement.

CHERI also changes MIPS's exception mechanism. Exception handlers execute privileged instructions and use memory that should not be accessible by other code. In MIPS, this is achieved by elevating the processor mode to kernel mode when an exception occurs. In CHERI-MIPS, this no longer suffices, so it also makes capabilities accessible that are normally not accessible. These capabilities are stored in special capability registers such as the kernel code capability (KCC) and the kernel data capability (KDC), which are only accessible if the PCC has the SystemRegisterAccess permission. When an exception occurs, the PCC is copied to the exception program counter capability (EPCC) and the KCC is copied to the PCC. Typically, the KCC has the SystemRegisterAccess permission, which then also makes the KDC accessible. The KCC and KDC usually have authority to

a region of memory that the original PCC does not have authority to, enabling the exception handler to store private data. On exception return, the EPCC is copied back to the PCC. Provided that this capability does not have the SystemRegisterAccess permission, the KCC and KDC are then no longer accessible.

Another change to the exception mechanism is the definition of new exception priorities. For example, code should not be able to trigger an address translation exception for virtual addresses it does not have authority to, because this would leak information. Exceptions that indicate a lack of authority therefore take higher priority than address translation exceptions.

Finally, CHERI extends MIPS's virtual memory based protection. In MIPS, TLB entries indicate whether the page can be read from, written to, and/or executed. CHERI-MIPS adds two new permissions, respectively whether valid capabilities can be read from and stored to the page. Cornucopia (see Section 1.3.5) adds status bits to TLB entries that makes its sweeping revocation algorithm more efficient.

2.2.3 New CHERI instructions

CHERI-MIPS defines 69 new instructions. Some of these implement the operations of CHERI's capability system that we described in Section 1.3.3, such as memory accesses that explicitly use the authority of a capability, and operations on capabilities, such as sealing, unsealing, and invoking them. Others are capability variants of MIPS instructions, such as conditional branches that use capability registers in their condition. Finally, there are instructions for common idioms such as converting a capability to an integer pointer.

CHERI-MIPS implements the new instructions with a MIPS *coprocessor*. MIPS is a modular architecture that supports up to four coprocessors: CP0 controls system registers, CP1 is an optional floating-point processor, and CP2 and CP3 are implementation defined. CHERI-MIPS uses CP2 to implement the new instructions, which we summarise below.

Loads and stores CHERI-MIPS adds load and store instructions that take an index of a general purpose capability register (GPCR) as parameter, and use the capability in that GPCR as authority. This explicit choice of authority follows the principle of intentional use, as explained in Section 1.3.1, and contrasts with legacy MIPS load and stores that implicitly use the authority of the DDC.

Some instruction variants preserve tags: they load a capability-sized and -aligned region of memory and its associated tag to a GPCR, or, similarly, store from a GPCR to memory. Other instruction variants can only be used for data: they load 8/16/32/64-bit words from memory to a GPR, or store them from a GPR to memory. Recall that storing data to memory clears the tag associated with the region of memory stored to.

Register copies CHERI-MIPS adds instructions that copy capabilities between GPCRs. There are conditional and unconditional variants. There are also instructions that copy capabilities between GPCRs and special capability registers such as the DDC, KCC, and KDC. As explained above, these instructions can only be executed if the PCC has the SystemRegisterAccess permission. Finally, there are instructions that copy the PCC to a GPCR. Copying in the other direction is possible through branches, which we explain below.

Branches There are new branch instructions that copy a capability from a GPCR to the PCC. The CJR instruction only updates the PCC, and CJALR also saves the original PCC in a GPCR. Like branches in MIPS, the effects of these instructions are delayed by one instruction. There are also new conditional branches that use a capability in the condition: CBEZ and CBNZ respectively test whether the capability is or is not the null capability (whose byte representation consists of zeroes), and CBTS and CBTU test whether the capability is respectively valid or invalid. These instruction only update the PC and leave the PCC unchanged.

Reading capability fields CHERI-MIPS adds instructions that copy a field of a capability to a GPR. The field can be the tag, address, base, length, permissions, sealed-bit, or object type, and the source capability must be in a GPCR. There are also instructions that assert that a field has a certain value: if the assertion is successful, the instruction has no effect, and otherwise it raises an exception. Finally, there are instructions for common computations on capability fields. For example, CGetOffset computes the offset between the base and the address of a capability, CSub computes the difference between the addresses of two capabilities, CPtrCmp compares the tags and addresses of two capabilities, and CToPtr computes the offset of the address of one capability to the base of another, provided both are valid capabilities.

Changing capability fields There are several instructions that change the fields of a capability. They all take a capability in a GPCR as input, copy it to a GPCR, and then change the fields of that copy. The fields can be changed in the following ways.

- The address can be changed with CSetAddr, CSetOffset, CFromPtr, and their variants. In versions of CHERI-MIPS that compress capabilities in memory, the new address must be representable, which limits how far the address may lie outside the bounds of the capability.
- Permissions can be removed with the CAndPerm instruction. The resulting permissions equal the bitwise AND of the original permissions and the instruction parameter. There are no instructions that add permissions.

- The bounds of a capability can be shrunk with the CSetBounds instruction and its variants. These change the base and length at the same time. They raise an exception if their input parameter specifies bounds that are larger than the original bounds of the capability. Like the address of a capability, the new bounds must be representable if a compression scheme is used.
- The tag of a capability can be cleared with CClearTag and its variants. The former clears the tag of one capability and its variants several at once.
- The CBuildCap instruction performs a combination of the instructions above. It takes a capability *cap* as input that may be invalid and/or sealed, and a capability *auth* that must be valid and unsealed. It creates a copy of *auth* and changes the address, base, length, and permissions of the copy to those of *cap*, if possible. These changes are not possible if they would increase the bounds or permissions of the copy. An alternative way of describing the instruction is that it creates a copy of *cap*, sets its tag, and unseals it, under the condition that *auth* has enough authority.
- The object type and sealed-bit can be changed by instructions that seal, unseal, or invoke capabilities. These are explained below.

Sealing and unsealing capabilities The CSeal and CUnseal instructions respectively seal and unseal capabilities. They take three GPCR register indices as parameter: one for the capability *auth* that is used as authority, one for the capability *cap* that is being (un)sealed, and one to write the result to. The address of *auth* determines the object type that CSeal uses to seal *cap* with. The instructions follow the requirements laid down in Section 1.3.3, which we repeat here. To seal *cap*, *auth* needs to be valid, unsealed, have the PermitSeal permission, and its address must lie in its memory region. To unseal a capability that has object type t, a similar condition holds: *auth* needs to be valid, unsealed, have the PermitUnseal permission, and t must lie in its memory region.

Capability invocation The CCallFast instruction invokes capabilities. It takes two GPCR register indices as parameter: one for the code capability and one for the data capability. CCallFast also follows the requirements laid down in Section 1.3.3, which we repeat here. The code capability must have the PermitExecute permission and its address must lie within its bounds. The data capability must not have the PermitExecute permission. Both capabilities need to be valid, have the PermitCCall permission, and be sealed with the same object type. If these are met, then CCallFast copies the code and data capabilities to respectively the PCC and GPCR 26, and unseals these copies. For this reason, GPCR 26 is sometimes called the invoked data capability (IDC). Unlike branches, the change of the PCC takes immediate effect.

The CCall instruction has a similar purpose, but does not invoke capabilities itself. It checks the same requirements as CCallFast, but then raises a particular exception that

system software is meant to handle. This is more flexible than CCallFast, as the exception handler could implement the invocation in various ways, but CCallFast performs better, as the name suggests.

Miscellaneous There are instructions to read from and write to the new exception status registers that CHERI-MIPS defines.

2.3 The L3 specification of CHERI-MIPS

L3 is a domain specific language for architecture specifications [55]. It is a strongly typed, first-order imperative language that is executable, that exports to interactive proof assistants, and that aims to be accessible to engineers without a formal background. The benefits of a formal specification over a prose specification were discussed in Section 1.4.1; here we describe the L3 specification of CHERI-MIPS [81] in more detail.

The L3 specification of CHERI-MIPS defines 197 instructions and around 350 auxiliary definitions in 7k non-comment lines of specification. The specification covers the entire architecture, including exceptions, the programmable interrupt controller (PIC), and the JTAG UART interface. This, combined with the fact that L3 is executable, makes it possible to boot FreeBSD on the specification.

L3 uses a state transition system to capture the semantics of architectures. Below we describe how the CHERI-MIPS state and its transition function are defined. We do this through examples, namely by showing parts of the specification of an instruction that can raise exceptions, one that can cause unpredictable behaviour, one that accesses memory, and one that manipulates a capability. We do not include their full specifications because of their size; the complete specification of CHERI-MIPS can be found online [81]. It is not necessary to understand all the details in the included parts. Their purpose is just to show how the semantics of CHERI-MIPS is specified.

2.3.1 Types

Types in L3 are built from the following primitives: `unit`, `bool`, `string`, `nat`, `int`, `bitstring`, and `bits(n)`. Types can be combined in function, sum, and product types. A *record type* is a special product type where each child type is named. For example, the MIPS system register is defined as a record type in Figure 2.1 on the following page. Then a *register type* is a record type that has a bijection with machine words. The name could be confusing: architectural registers are typically specified with a register type, but this is not a requirement, as we saw above; and vice versa, register types can be used to specify other things than architectural registers. For example, capabilities are defined as a register type in Figure 2.2 on page 54.

```
record CP0
{
    Index     :: Index           -- Index to TLB array
    Random    :: Random          -- Pseudorandom pointer to TLB array
    EntryLo0  :: EntryLo         -- Low half of TLB entry for even VPN
    EntryLo1  :: EntryLo         -- Low half of TLB entry for odd VPN
    Context   :: Context         -- Kernel virtual page table entry (PTE)
    UsrLocal  :: bits(64)        -- UserLocal register
    PageMask  :: PageMask        -- TLB page mask
    Wired     :: Wired           -- Number of wired TLB entries
    HWREna    :: HWREna          -- See RDHWR instruction
    BadVAddr  :: bits(64)        -- Bad virtual address
    BadInstr  :: bits(32)        -- Instruction cause of the exception
    BadInstrP :: bits(32)        -- Branch before exception cause
    Count     :: bits(32)        -- Timer count
    EntryHi   :: EntryHi         -- High half of TLB entry
    Compare   :: bits(32)        -- Timer compare
    Status    :: StatusRegister  -- Status register
    Cause     :: CauseRegister   -- Cause of last exception
    EPC       :: bits(64)        -- Exception program counter
    PRId      :: bits(32)        -- Processor revision identifier
    Config    :: ConfigRegister  -- Configuration register
    Config1   :: ConfigRegister1 -- Configuration register 1
    Config2   :: ConfigRegister2 -- Configuration register 2
    Config3   :: ConfigRegister3 -- Configuration register 3
    Config6   :: ConfigRegister6 -- Configuration register 6
    LLAddr    :: bits(64)        -- Load linked address
    XContext  :: XContext        -- PTE entry in 64-bit mode
    Debug     :: bits(32)        -- EJTAG Debug register
    ErrCtl    :: bits(32)        -- Error Control
    ErrorEPC  :: bits(64)        -- Error exception program counter
}
```

Figure 2.1: The L3 specification of the MIPS system register, which is controlled by coprocessor 0. This register stores the status of the processor, including its operating mode and exception level; it stores information necessary to handle exceptions, such as the cause, the last instruction, and, in case of address translation exceptions, the relevant virtual address and TLB entries; it stores the configuration of the processor; and various other things. For the specification of the types of the fields, we refer to the L3 source [81].

```
register Capability :: bits(257)
{
    256     : tag      -- Whether it is valid
    255-192 : length   -- Length of the memory region
    191-128 : base     -- Base of the memory region
    127-64  : cursor   -- Address
    63-56   : reserved -- Reserved for future use
    55-32   : otype    -- Object type
    31-16   : uperms   -- User permissions
    15-1    : perms    -- Permissions
    0       : sealed   -- Whether it is sealed
}
```

Figure 2.2: The L3 specification of capabilities in the version of CHERI-MIPS that uses uncompressed, 256-bit capabilities. The type `Capability` is a record: `tag` and `sealed` are booleans and the other fields are machine words whose lengths are given by the specified ranges. L3 automatically generates a bijection between `Capability` and 257-bit words based on these ranges. For example, the 257th bit corresponds to the tag.

2.3.2 Architectural state

Conceptually, the architectural state is a record type, but L3 does not require it to be defined in one place. Instead, the state is built throughout the specification with statements of the form `declare foo :: t`, which adds a field `foo` of type `t` to the state. For example, the specification in Figure 2.3 on the following page adds several capability registers to the state.

L3 also supports *components* that can be read from and written to: they consist of a *getter* function that returns a value, and a *setter* function that has a value as input and returns `unit`. Components are typically used to wrap state fields. For example, the component `CAPR` in Figure 2.4 on the next page wraps around `c_capr` to ensure that register 0 always contains the null capability.

All the architectural state of CHERI-MIPS is explicitly defined using fields and components, including state that was left implicit in the prose specification. For example, the prose specification states that the effects of branches is delayed with one instruction, but the L3 specification captures this more precisely: branches leave `PC` and `PCC` unchanged, but update `BranchDelay` or `BranchDelayPCC`. These are copied to respectively `PC` or `PCC` when the instruction in the branch delay slot has finished executing. We summarise the most relevant fields and components in Figure 2.5 on page 57. Note that some architectural state is captured by subfields of these. For example, the EPC is captured as `CP0.EPC` and the exception level as `CP0.Status.EXL`.

The L3 specification also adds fields and components that do not correspond to architectural state. We call this *ghost state*. For example, `BranchTo` and `BranchToPCC` are internally used in the specification of branches, but are not part of the CHERI-MIPS architecture, and therefore do not need to be implemented in conforming hardware implementations.

```
declare
{
    c_pcc   :: Capability            -- Program counter capability
    c_capr  :: bits(5) -> Capability -- General purpose capability registers
    c_scapr :: bits(5) -> Capability -- Special capability registers
}
```

Figure 2.3: The L3 specification that adds the program counter capability (PCC), the general purpose capability registers (GPCRs), and the special capability registers (SCRs) to the state of CHERI-MIPS. There are 32 GPCRs and 32 SCRs. Both are specified as functions from 5-bit machine words to capabilities. Figure 2.5 on the following page gives an overview of other parts of the state that are relevant in this thesis.

```
component CAPR (n::reg) :: Capability
{
    -- Register 0 always returns the null capability
    value = if n == 0 then nullCap else c_capr(n)
    assign value =
    {
        -- Writing to register 0 is allowed, but it does
        -- not change the value that is read from CAPR 0
        c_capr(n) <- value;
        -- Log the access
        mark_log (2, log_creg_write (n, value))
    }
}
```

Figure 2.4: The L3 specification of the `CAPR` component that ensures that general purpose capability register 0 (GPCR 0) always contains the null capability, and that updates to the registers are logged. Instructions are meant to access GPCRs through `CAPR` and not through `c_capr`, but L3 does not enforce this.

Name	Description
`PC`	Program counter.
`PCC`	Program counter capability.
`BranchDelay`	If the current instruction is in a branch delay slot, `BranchDelay` is the PC that the branch jumped to. Otherwise, it is empty.
`BranchDelayPCC`	If the current instruction is in a branch delay slot, `BranchDelayPCC` is the PCC that the branch jumped to. Otherwise, it is empty.
`BranchTo`	If the current instruction is a branch, `BranchTo` is the PC that is jumped to. Otherwise, it is empty.
`BranchToPCC`	If the current instruction is a branch, `BranchToPCC` is the PCC that is jumped to. Otherwise, it is empty.
`MEM`	Physical memory.
`GPR`	General purpose registers.
`CAPR`	General purpose capability registers.
`IDC`	Invoked data capability. Alias of `CAPR(26)`.
`SCAPR`	Special capability registers.
`DDC`	Default data capability. Alias of `SCAPR(0)`.
`KCC`	Kernel code capability. Alias of `SCAPR(29)`.
`KDC`	Kernel data capability. Alias of `SCAPR(30)`.
`EPCC`	Exception program counter capability. Alias of `SCAPR(31)`.
`CP0`	MIPS system registers. The type `CP0` is defined in Figure 2.1 on page 53.
`currentInst`	The encoding of the current instruction, if known.
`lastInst`	The encoding of the previous instruction, if known.
`exceptionSignalled`	Indicates whether the current instruction raised an exception.
`capcause`	Capability exception cause register.

Figure 2.5: An overview of state fields and state components in the L3 specification of CHERI-MIPS. We included the fields and components that are most relevant to this thesis. For the remaining fields and components, we refer to the L3 source [81].

2.3.3 Hardware exceptions

CHERI changes MIPS's exception mechanism, as explained in Section 2.2.2. The new exception mechanism is captured by the auxiliary function `SignalException`, whose L3 specification is given in Figure 2.7 on the following page. This specification is a function from `ExceptionType` to `unit` that can read and change the architectural state. For example, `CP0.EPC <- PC` at Line 24 on the next page reads the `PC` state component and copies it to the the `EPC` field of the `CP0` state component. This specification shows details that we skimmed over before, namely that the behaviour of hardware exceptions is different if another exception is already being handled. The exception level `CP0.Status.EXL` indicates whether this is the case.

Instructions that raise a hardware exception call `SignalException` either directly or through other auxiliary functions. For example, the DADD instruction in Figure 2.6 calls `SignalException` directly when its operands overflow. Other instructions may call auxiliary functions that store more information about the exception. For example, if an address `vAddr` cannot be translated, the auxiliary function `SignalTLBException` is called, which (among other things) saves the address to `CP0.BadVAddr` and then calls `SignalException`.

```
-- The "double add" instruction
define DADD (rs::reg, rt::reg, rd::reg) =
{
    -- Perform a 65-bit addition
    temp‘65 = SignExtend(GPR(rs)) + SignExtend(GPR(rt));
    -- Check for overflow
    if temp<64> <> temp<63> then
        -- Raise an overflow exception
        SignalException(Ov)
    else
        -- Store the 64-bit result
        GPR(rd) <- temp<63:0>
}
```

Figure 2.6: The L3 specification of DADD that adds two signed 64-bit values. It extends the 64-bit operands `GPR(rs)` and `GPR(rt)` to 65-bit and performs a 65-bit addition. If the two most significant bits of the result do not have the same value, the addition overflowed and the instruction raises a hardware exception. Otherwise, it truncates the result to 64 bits and writes it to `GPR(rd)`. The specification of `SignalException` is given in Figure 2.7 on the following page.

```
1  unit SignalException (ExceptionType::ExceptionType) =
2  {
3      -- Give BadInstrP an unpredictable value
4      CP0.BadInstrP <- UNKNOWN(next_unknown("BadInstrP"));
5      -- Check whether another exception is being handled
6      when not CP0.Status.EXL do
7      {
8          -- Check whether there is a pending branch
9          if IsSome(BranchDelay) or IsSome(BranchDelayPCC) then
10         {
11             mark_log(2, "EPC <- ":hex (PC - 4):
12                         " (in branch delay slot => PC - 4 )");
13             -- Copy the address of the branch to the EPC
14             CP0.EPC <- PC - 4;
15             -- Set BD to state that a branch was pending
16             CP0.Cause.BD <- true;
17             -- Store the branch instruction (if it exists)
18             when IsSome(lastInst) do CP0.BadInstrP <- ValOf(lastInst)
19         }
20         else
21         {
22             mark_log(2, "EPC <- ":hex(PC));
23             -- Copy the PC to the EPC
24             CP0.EPC <- PC;
25             -- Clear BD to state that no branch was pending
26             CP0.Cause.BD <- false
27         }
28     };
29     -- Determine the offset from the base address of the handler
30     vectorOffset =
31     if (ExceptionType == XTLBRefillL or ExceptionType == XTLBRefillS) and
32         not CP0.Status.EXL then 0x080‘30
33     else if (ExceptionType == C2E and
34             (capcause.ExcCode == 0x5 or capcause.ExcCode == 0x6)) then 0x280
35     else 0x180;
```

(Continued on the next page)

Figure 2.7: The L3 specification of `SignalException`, an auxiliary function that is called whenever a hardware exception is raised. It saves the correct address to the EPC (Lines 6–28) and computes an offset that is used to determine the address of the exception handler (Lines 30–35). This is page 1 of 2 of the figure.

```
36      -- Store the instruction that caused the exception (if it exists)
37      when IsSome(currentInst) do CP0.BadInstr <- ValOf(currentInst);
38      -- Store the type of exception
39      CP0.Cause.ExcCode <- ExceptionCode(ExceptionType);
40      -- Determine the base address of the handler
41      vectorBase = if CP0.Status.BEV then 0xFFFF_FFFF_BFC0_0200‘64
42                   else 0xFFFF_FFFF_8000_0000;
43      -- Clear any pending branches
44      BranchDelay    <- None;
45      BranchTo       <- None;
46      BranchDelayPCC <- None;
47      BranchToPCC    <- None;
48      -- Indicate that an exception has occurred. The flag is not
49      -- architectural state, but can be used by other L3 definitions.
50      exceptionSignalled <- true;
51      -- Copy PCC to EPCC, unless another exception is being handled
52      var new_epcc = PCC;
53      if not canRepOffset(PCC, PC) then
54          new_epcc <- setOffset(nullCap, getBase(PCC) + PC)
55      else
56          new_epcc <- setOffset(new_epcc, PC);
57      when not CP0.Status.EXL do EPCC <- new_epcc;
58      -- Copy KCC to PCC
59      PCC <- KCC;
60      -- Jump to the handler
61      PC <- (vectorBase<63:30> : (vectorBase<29:0> + vectorOffset)) -
62            getBase(PCC);
63      -- Set the exception level
64      CP0.Status.EXL <- true;
65      -- Log the exception
66      mark_log(2, log_sig_exception(ExceptionCode(ExceptionType)))
67  }
```

Figure 2.7: This is page 2 of 2 of the figure. The specification shown here stores the exception type (Line 39), determines the base address of the exception handler (Lines 41–42), cancels any pending branches (Lines 44–47), copies the PCC to the EPCC and the KCC to the PCC (Lines 52–59), jumps to the exception handler (Line 61), and sets the exception level (Line 64).

2.3.4 Unpredictable behaviour

In some corner cases, the MIPS architecture specifies that an instruction has unpredictable behaviour, as described in Section 2.1. The L3 specification of CHERI-MIPS only partly formalises this: it specifies when unpredictable behaviour occurs, but it does not specify the effects of unpredictable behaviour, as mentioned in Section 1.5.3. The reason is that L3 can only specify deterministic semantics.

The L3 specification uses L3 exceptions to specify when unpredictable behaviour occurs. It first defines the exception with `exception UNPREDICTABLE :: string`, and then uses `#UNPREDICTABLE("Some info")` to raise the exception. For example, the specification of ADDU in Figure 2.8 on the next page raises this L3 exception if the operands of the addition cannot be interpreted as 32-bit values.

L3 exceptions should not be confused with hardware exceptions. From the viewpoint of L3, there is no fundamental difference between hardware exceptions and other architectural behaviour. They are fully specified, can be executed, and have a formal semantics. An L3 exception, on the other hand, means that there is no semantics for that situation. When executing an L3 specification, L3 exceptions abort the execution. When exporting to automated proof assistants, L3 exceptions are represented by flags in the architectural state: if one of the flags is set, the corresponding L3 exception has been raised, and the rest of the state is meaningless.

Specifying unpredictable behaviour with L3 exceptions is sufficient when developing the software stack above CHERI-MIPS: the software stack should not rely on unpredictable behaviour, so one only needs to know when it occurs. However, we need to know the semantics of unpredictable behaviour to be able to check whether our properties hold. We therefore formalise the semantics ourselves in Section 2.5.4.

```
-- The "add unsigned" instruction
define ADDU (rs::reg, rt::reg, rd::reg) =
{
    -- Check whether the operands can be interpreted as 32-bit values
    -- If either cannot, the behaviour is unpredictable
    when NotWordValue(GPR(rs)) or NotWordValue(GPR(rt))
    do #UNPREDICTABLE("ADDU: NotWordValue");
    -- Perform a 32-bit addition
    temp = GPR(rs)<31:0> + GPR(rt)<31:0>;
    -- Extend to 64-bit by extending the sign and store the result
    GPR(rd) <- SignExtend(temp)
}
```

Figure 2.8: The L3 specification of ADDU that adds two unsigned 32-bit values. The operands of the addition are the register values `GPR(rs)` and `GPR(rt)`, which are 64-bit. To check whether these can be interpreted as 32-bit values, the specification uses the auxiliary function `NotWordValue`. If this is not the case, the behaviour is unpredictable. Otherwise, it performs the 32-bit addition, extends the sign of the result to form a 64-bit value, and writes that to `GPR(rd)`.

2.3.5 Memory accesses

CHERI-MIPS's tagged memory is specified by dividing the memory into capability-sized and -aligned regions. The sizes that we mention here are from the version that does not compress capabilities in memory. To access a byte in memory, one first uses the upper 35 bits of a physical address to determine the region, and then the lower 5 bits to determine the byte within the region. This is specified in L3 as follows. The `MEM` state component is a function from `bits(35)` to `DataType`, and `DataType` is a tagged union type with the constructors `Cap` that takes a `Capability`, and `Raw` that takes a `bits(256)`. The tag of a region represented by `Cap cap` equals the tag of `cap`, and the tag of a region represented by `Raw x` is unset. To access bytes within a region represented by `Cap cap`, one converts `cap` with the auxiliary function `capToBits`, and, vice versa, to access a region represented by `Raw x` as a capability, one converts `x` to an (invalid) capability with `bitsToCap`.

CHERI-MIPS's address translation is specified as the function `AddressTranslation`, which takes a virtual address as input, and returns the translated address, the cacheability and coherency attributes, and whether the TLB allows storing and/or loading capabilities here. `AddressTranslation` calls `CheckSegment` to see whether the virtual address lies in a segment that is accessible in the current processor mode, and if so, whether the address is mapped or unmapped. If the address is mapped, it calls `LookupTLB` to find matching TLB entries. At various points, `AddressTranslation` and its supporting auxiliary functions can raise exceptions and cause unpredictable behaviour. We described address translation in more detail in Section 2.1.2.

Then there are various auxiliary functions that load from and store to memory. For example, `LoadCap` takes a virtual address as input and a boolean that indicates whether the load is linked, and it returns a capability. It translates the address, checks whether the address is in use by the UART or a PIC, checks whether the TLB allows loading capabilities from this address, and loads the capability. If the TLB does not allow loading capabilities or if the memory region contains data, the function returns an invalid capability. The specification of `LoadCap` is shown in Figure 2.9 on the following page. Similar auxiliary functions are `LoadMemoryCap` that loads aligned or unaligned data, `StoreCap` and `StoreMemoryCap` that respectively store capabilities and data, and `Fetch` that loads an instruction from memory.

Instructions that access memory use these auxiliary methods. For example, the `CLC` instruction loads a capability from memory: it checks whether the capability that is used as authority indeed has enough authority, and if so, it calls `LoadCap` to load the capability. If the capability that is used as authority has the PermitLoad permission, but not PermitLoadCapability, it removes the tag of the loaded capability. Its specification is shown in Figure 2.10 on page 65. Other instructions that access memory, including legacy memory accesses, are specified in a similar way.

```
Capability LoadCap (vAddr::vAddr, link::bool) =
{
    -- Translate the virtual address. pAddr is the physical address,
    -- CCA the cacheability and coherency attribute, and L indicates
    -- whether the TLB entry forbids loading capabilities from pAddr.
    pAddr, CCA, _, L = AddressTranslation (vAddr, LOAD);
    if exceptionSignalled then UNKNOWN(next_unknown("capability"))
    else
    {
        a = pAddr<39:Log2(CAPBYTEWIDTH)>;
        -- Check whether the address is in use by the UART or a PIC
        -- If so, the behaviour is unpredictable
        if a == JTAG_UART.base_address<36:capbottom> then
            #UNPREDICTABLE ("Capability load attempted on UART")
        else for core in 0 .. totalCore - 1 do
        {
           base = PIC_base_address([core]);
           when base<36:capbottom> <=+ a and
                a <+ (base+1072)<36:capbottom>
           do #UNPREDICTABLE ("Capability load attempted on PIC")
        };
        -- If the load is linked, save the linked address
        if link then
        {
            when CCA == 2
            do #UNPREDICTABLE("load linked on uncached address");
            LLbit <- Some (true);
            CP0.LLAddr <- [pAddr]
        }
        else LLbit <- None;
        -- Retrieve the capability from memory
        var cap = ReadCap(a);
        -- Remove the tag if we are not allowed to load capabilities
        when L do cap <- setTag(cap, false);
        memAccessStats.bytes_read <- memAccessStats.bytes_read +
                                     CAPBYTEWIDTH;
        mark_log (2, "Load cap: " : log_load_cap (pAddr, cap) :
                     " from vAddr ":hex (vAddr));
        watchForCapLoad(pAddr, cap);
        return cap
    }
}
```

Figure 2.9: The L3 specification of `LoadCap`, an auxiliary function that loads a capability from memory. Among other things, it checks whether the capability's address is in use by the UART of a PIC and whether the TLB allows loading capabilities from the address.

```
1  -- The "load capability" instruction
2  define CLC (cd::reg, cb::reg, rt::reg, offset::bits(11)) =
3  {
4      -- Check whether the capability coprocessor is enabled
5      if not CP0.Status.CU2 then SignalCP2UnusableException
6      -- Check whether the capability that is used as authority is valid,
7      -- unsealed, and has permission to load
8      else if not getTag(CAPR(cb)) then SignalCapException(capExcTag,cb)
9      else if getSealed(CAPR(cb)) then SignalCapException(capExcSeal,cb)
10     else if not getPerms(CAPR(cb)).Permit_Load then
11         SignalCapException(capExcPermLoad,cb)
12     else
13     {   -- Determine the virtual address that is loaded from
14         cursor = getBase(CAPR(cb)) + getOffset(CAPR(cb));
15         extOff = offset:'0000';
16         addr   = cursor + GPR(rt) + SignExtend(extOff);
17         -- Check whether cap cb has authority over this address
18         if ('0':addr) + [CAPBYTEWIDTH] >+
19            ('0':getBase(CAPR(cb))) + ('0':getLength(CAPR(cb))) then
20             SignalCapException(capExcLength,cb)
21         else if addr <+ getBase(CAPR(cb)) then
22             SignalCapException(capExcLength,cb)
23         -- Check whether the virtual address is aligned
24         else if not isCapAligned([addr]) then
25         {
26             CP0.BadVAddr <- [addr];
27             SignalException(AdEL)
28         }
29         else
30         {   -- Load the capability from memory
31             var tmp = LoadCap([addr], false);
32             -- Check whether cap cb has permission to load capabilities
33             when not getPerms(CAPR(cb)).Permit_Load_Capability
34             -- If not, make the loaded capability invalid
35             do tmp <- setTag(tmp, false);
36             -- Store the loaded capability in the destination register
37             when not exceptionSignalled do CAPR(cd) <- tmp;
38             LLbit <- None
39         }
40     }
41 }
```

Figure 2.10: The L3 specification of CLC that loads a capability from memory. It checks whether the capability that is used as authority indeed has enough authority to load the capability, and if so, it calls the auxiliary function `LoadCap` (Line 31), whose specification is given in Figure 2.9 on the previous page.

2.3.6 Capability manipulations

The L3 specification defines auxiliary functions to modify the fields of a capability: `setTag`, `setType`, `setPerms`, `setUPerms`, `setSealed`, `setOffset`, and `setBounds`. The definition of these functions depends on whether the version of the L3 specification compresses capabilities in memory or not, but their signature is always the same. Instructions only manipulate capabilities through these functions, and have the same specification in all versions of the L3 specification.

Instructions that manipulate capabilities check whether the manipulation is allowed. For example, the `CUnseal` instruction checks whether the capability that is used as authority is valid, is unsealed, has the PermitUnseal permission, and contains the relevant object type in its memory region. If this is the case, `CUnseal` uses the `setSealed`, `setType`, and `setPerms` functions to unseal the source capability. The specification of `CUnseal` is shown in Figure 2.11 on the following page.

2.3.7 The entire execution step

The semantics of an entire execution step is captured by the function `Next`. It calls `Fetch` to fetch the next instruction from memory, decodes it, and then calls `Run` to execute it. Similar to the architectural state, the function `Run` is not defined in one place, but throughout the L3 specification: it is a case split over all the instruction specifications. The `define` keyword indicates whether a function is an instruction or an auxiliary function. After calling `Run`, `Next` checks whether the current and/or previous instruction was a branch. If both of them were, the behaviour is unpredictable; if only the current instruction was a branch, it copies `BranchTo` and `BranchToPCC` to respectively `BranchDelay` and `BranchDelayPCC`; and if only the previous instruction was a branch, it copies `BranchDelay` and `BranchDelayPCC` to respectively `PC` and `PCC`. Except in the last case, the `PC` is increased by 4.

To execute a program on the L3 specification, one first defines an L3 state that, among other things, contains the program in its memory and whose PC points to the program. By calling `Next`, this state is transformed to the state that is the result of executing one instruction, so by repeatedly calling `Next` one can execute the program.

In theorem prover exports, `Next` is transformed to a function from states to states that captures the semantics of CHERI-MIPS. We describe the Isabelle/HOL export in the next section.

```
1  -- The "unseal capability" instruction.
2  define CUnseal (cd::reg, cs::reg, ct::reg) =
3  {
4      -- Check whether the capability coprocessor is enabled
5      if not CP0.Status.CU2 then
6          SignalCP2UnusableException
7      -- Check whether the source (the capability in cs) is valid and sealed,
8      -- and the authority (the capability in ct) is valid and unsealed
9      else if not getTag(CAPR(cs)) then
10         SignalCapException(capExcTag,cs)
11     else if not getTag(CAPR(ct)) then
12         SignalCapException(capExcTag,ct)
13     else if not getSealed(CAPR(cs)) then
14         SignalCapException(capExcSeal,cs)
15     else if getSealed(CAPR(ct)) then
16         SignalCapException(capExcSeal,ct)
17     -- Check whether the address of the authority equals the
18     -- object type of the source
19     else if (getBase(CAPR(ct)) + getOffset(CAPR(ct))) <>
20              ZeroExtend(getType(CAPR(cs))) then
21         SignalCapException(capExcType,ct)
22     -- Check whether the authority has permission to unseal
23     else if not getPerms(CAPR(ct)).Permit_Unseal then
24         SignalCapException(capExcPermUnseal,ct)
25     -- Check whether the address of the authority is valid
26     else if getOffset(CAPR(ct)) >=+ getLength(CAPR(ct)) then
27         SignalCapException(capExcLength,ct)
28     else
29     {
30         -- Create a copy of the source and unseal it
31         var new_cap = CAPR(cs);
32         new_cap <- setSealed(new_cap, false);
33         new_cap <- setType(new_cap, 0);
34         -- Remove the Global permission if the authority does not have it
35         var p::Perms = getPerms(new_cap);
36         p.Global <- getPerms(CAPR(cs)).Global and getPerms(CAPR(ct)).Global;
37         new_cap  <- setPerms(new_cap, p);
38         -- Store the resulting capability
39         CAPR(cd) <- new_cap
40     }
41 }
```

Figure 2.11: The L3 specification of CUnseal that copies and unseals a capability. It checks whether the requirements of unsealing capabilities described in Section 1.3.3 are met (Lines 7–27), and if so, it unseals the capability (Lines 30–39). Note that the resulting capability is only global if both the source and the authority are global (Lines 34–37).

2.4 The Isabelle/HOL export of CHERI-MIPS

Here we describe the Isabelle/HOL export of the L3 specification.

2.4.1 Notation

To make our definitions more accessible, we deviate from Isabelle/HOL's syntax. Normally, Isabelle terms are strictly separated from terms in the logic, which in our case is higher-order logic. We no longer show this separation, except in one case: we still distinguish Isabelle's meta-equality from equality in HOL. We use "≡" for the former, which should be read as "is defined as", and we use "=" for the latter.

We use `blue lower case with spaces` for keywords, `orangeCamelCase` for variables, and `BlackPascalCase` for named definitions. For accessibility, we treat boolean connectives and conditional statements as keywords: we write `for all` for universal quantification, `if then` for implications, `and` for conjunctions, et cetera. Note that the implication `if then` is a boolean, while the conditional statement `if then else` can be of any type.

As usual in Isabelle and in functional languages, we write function application with juxtaposition. For example, `Foo x` is the `Foo` function applied to `x`.

2.4.2 Algebraic data types

Isabelle supports tagged union types, and automatically generates constructors for them. For example, `None` of type `'a Option` and `Some` of type `'a ⇒ 'a Option` are the constructors of the `'a Option` type. Here, `'a` is a type variable that can be instantiated with any type. Isabelle also supports record types, and automatically generates projection and update functions for each field, collectively called field accessors. For example, consider the record `Point` with integer fields `X` and `Y`. Then `X` is a projection function of type `Point ⇒ Int`, and we write `p(X := x)` for the point whose `X` field equals `x`, and whose other fields equal those of `p`.

There is no fundamental difference between constructors, field accessors, and user defined functions. Tagged union types and record types are formalised respectively as sets [12] and composite tuples [104], and constructors and field accessors are simply definitions on these underlying types.

2.4.3 Machine words

It is desirable to use different types for machine words of different lengths. Isabelle does not support dependent types, but uses a "trick" to achieve this. The type of machine words is `'a Word`, where `'a` is a type whose cardinality `|'a|` is a natural number. It is defined through an embedding in integers, mapping to `0`, ..., `2^|'a| - 1`. Then there are

types `0`, `1`, `2`, ... that have a matching cardinality, which means we can write `8 Word` for the type of bytes, for example. Standard operations such as addition and the order over integers are lifted to machine words through this embedding. Note that the lifted order is the unsigned order over machine words.

Many well-known facts about integers do not hold for machine words. For example, it seems obvious that `a` $\leq$ `a + b` when `b` $\geq$ `0`, but this is not the case for `a = 0x01` and `b = 0xFF`. Their counter-intuitive behaviour contributed to several bugs in the L3 specification of CHERI-MIPS: we found a bug that allowed accessing the start of the address space if one had permission to the end of the address space; another bug allowed accessing memory one byte beyond the memory region one had permission to; and one bug gave access to the wrong region of memory when using unaligned accesses (see Chapter 6).

Despite their embedding in integers, we recommend thinking of an `'a Word` as an |`'a`|-tuple of booleans. Many operations on machine words such as shifts, concatenation, and bitwise AND, OR, and XOR are easy to understand with this mental model, and it might be more apparent that addition wraps.

In Figure 2.12 on the next page we summarise the auxiliary functions about machine words that we use in our security properties.

2.4.4 Capabilities

The L3 specification of the `Capability` type is exported to an Isabelle record. Our formal security properties interact with capabilities through the field accessors that Isabelle automatically generates, which are shown in Figure 2.13 on page 71. Note that we renamed the fields of the `Capability` type for readability: we use PascalCase, following our Isabelle notation, and we use more descriptive terms, changing `otype` to `ObjectType`, `cursor` to `Address`, `uperms` to `UserPerms`, and `sealed` to `IsSealed`.

Our formal security properties also use the offset of a capability. The offset is not a field, but a derived value, namely the difference between the base and the address of the capability. Its formal definition is straightforward: `Offset cap` $\equiv$ `Address cap - Base cap`. Note that the subtraction wraps if `Address cap < Base cap`. For each system permission, we define an auxiliary function that retrieves the corresponding bit from the `Perms` field of the capability, as shown in Figure 2.14 on page 72.

In the PDF version of this thesis, many definitions are clickable and link to (roughly) the place where they are defined. For example, `Address` links to Figure 2.13 on page 71 and `Offset` links to this subsection.

`w AND u`, `w OR u`, and `w XOR u`
: Returns respectively the bitwise AND, OR, and XOR of the words `w` and `u`.

`Bit w n`
: Returns the `n`-th bit of `w` as a boolean. If `n` is greater than or equal to the size of `w`, it returns `False` instead.

`ExtractByte n w`
: Returns the bits `8*n`, ..., `8*n + 7` of `w` as an `8 Word`.

`ExtractWord n m w`
: Returns the bits `m`, ..., `n` of `w` as an `'a Word`. It is meant to be used with `|'a| = n - m + 1`, but Isabelle does not support dependent types, so this cannot be enforced. If `|'a|` $\neq$ `n - m + 1`, it returns the unsigned cast of the extracted bits.

`Mask n`
: Returns an `'a Word` whose least significant `n` bits are set.

`SignedCast w`
: Casts an `'a Word` to a `'b Word`. If `|'a| < |'b|`, then the sign of `w` is repeated as padding. If `|'a| > |'b|`, the sign of `w` becomes the sign of the result, truncating the `|'a| - |'b|` bits after the sign.

`Size w`
: Returns the size of `w`, namely `|'a|` if `w` is of type `'a Word`.

`Slice n w`
: Removes the `n` least significant bits from `w` of type `'a Word` and returns this as a `'b Word`. It is meant to be used with `|'b| = |'a| - n`, but this cannot be enforced. If `|'b|` $\neq$ `|'a| - n`, it returns the unsigned cast of the slice.

`UnsignedCast w`
: Casts an `'a Word` to a `'b Word`. If `|'a| < |'b|`, then `w` is padded with zeroes. If `|'a| > |'b|`, the most significant bits of `w` are truncated.

`WordCat w u`
: Concatenates `w` of type `'a Word` and `u` of type `'b Word` to a `'c Word`. It is meant to be used with `|'c| = |'a| + |'b|`, this cannot be enforced. If `|'c|` $\neq$ `|'a| + |'b|`, it returns the unsigned cast of the concatenation.

`WordToNat w`
: The word `w` is converted to a natural number. The range of the function is `0`, ..., $2^{|'a|}$ `- 1` for words of type `'a Word`.

Figure 2.12: Auxiliary functions about machine words that we use to define our security properties. For their formal definitions, we refer to our Isabelle development.

Name	Type	Description
`Address cap`	`64 Word`	Returns the address `cap` points to.
`Base cap`	`64 Word`	Returns the start of the memory region that `cap` has authority to.
`IsSealed cap`	`Bool`	Returns whether `cap` is sealed.
`Length cap`	`64 Word`	Returns the length of the memory region that `cap` has authority to.
`ObjectType cap`	`24 Word`	Returns the object type of `cap`.
`Perms cap`	`15 Word`	Each bit in the returned word specifies whether `cap` has the corresponding system permission (see Figure 2.14 on the next page).
`Reserved cap`	`8 Word`	Returns the part of `cap` that is reserved for future use.
`Tag cap`	`Bool`	Returns whether `cap` is valid.
`UserPerms cap`	`16 Word`	Each bit in the returned word specifies whether `cap` has the corresponding user-defined permission.

Figure 2.13: The projection functions that Isabelle generates for the `Capability` record type. For each of the shown fields, there is also a function that updates that field. For example, `cap(Tag := x)` returns a capability that equals `cap`, except its tag equals `x`. Note that we renamed some of the fields for readability, as explained in Section 2.4.4.

`IsGlobal cap ≡ Bit (Perms cap) 0`

If this bit is set, the capability is global as opposed to local. This information flow label is described in Section 1.3.3.

`PermitExecute cap ≡ Bit (Perms cap) 1`

If this bit is set, `cap` can authorise instruction execution.

`PermitLoad cap ≡ Bit (Perms cap) 2`

If this bit is set, `cap` can authorise memory accesses that load data.

`PermitStore cap ≡ Bit (Perms cap) 3`

If this bit is set, `cap` can authorise memory accesses that store data.

`PermitLoadCapability cap ≡ Bit (Perms cap) 4`

If this bit is set and `PermitLoad cap` is true, `cap` can authorise memory accesses that load capabilities.

`PermitStoreCapability cap ≡ Bit (Perms cap) 5`

If this bit is set and `PermitStore cap` is true, `cap` can authorise memory accesses that store global capabilities.

`PermitStoreLocalCapability cap ≡ Bit (Perms cap) 6`

If this bit is set, and `PermitStore cap` and `PermitStoreCapability cap` are true, `cap` can authorise memory accesses that store local capabilities.

`PermitSeal cap ≡ Bit (Perms cap) 7`

If this bit is set, `cap` can authorise sealing capabilities.

`PermitCCall cap ≡ Bit (Perms cap) 8`

If this bit is set, `cap` can be invoked. The name is derived from the CCall instruction that invokes capabilities.

`PermitUnseal cap ≡ Bit (Perms cap) 9`

If this bit is set, `cap` can authorise unsealing capabilities.

`PermitAccessSystemRegisters cap ≡ Bit (Perms cap) 10`

If this bit is set, `cap` can authorise system register accesses.

Figure 2.14: The auxiliary functions that return which system permissions a capability possesses.

2.4.5 Machine state

The machine state, which is defined throughout the L3 specification, is exported to the Isabelle record `State`. This type is too large to include here: it has 59 fields, and transitively refers to 28 other record types, which have a combined total of 211 fields, so we describe only the most used fields.

We improved the readability of the machine state in two ways. First, we renamed some fields to avoid uncommon acronyms, for example renaming `RE` to `ReverseEndian`, but we still use common acronyms such as `GPR` (general purpose registers), `PC` (program counter), and `PCC` (program counter capability). We use a compromise for general purpose capability registers: `GPCR` is an uncommon acronym, but the entire name is too long, so we settled for `GPCapReg`. Second, we define abbreviations for nested subfields. For example, to retrieve the exception PC, one first has to get the state of the currently running processor core (`ProcessorState`), then its system registers (`CP0`), and finally the exception PC. We define an auxiliary function that returns the subfield immediately: `EPC s ≡ _EPC (CP0 (ProcessorState s))`. We prefix the original subfield with an underscore to distinguish it from the abbreviation. We also prefix fields that should not be accessed directly. For example, the auxiliary function `GPR` always returns 0 for register 0, and internally uses the state field `_GPR`.

The fields and abbreviations that are necessary to state our security properties are defined in Appendix A.1. Below we summarise the most important ones.

- `PCC s` returns the PCC, `GPCapReg s i` returns the `i`-th general purpose capability register, and `SpecialCapReg s i` returns the `i`-th special capability register. For some of these we define abbreviations: `InvokedDataCap s` returns the 26th general purpose capability register, and `DefaultDataCap s`, `KernelCodeCap s`, and `EPCC s` (the exception PCC) return respectively the 0th, 29th, and 31st special capability register.
- If the current instruction is in a branch delay slot, `BranchDelay s` returns the address that is jumped to, and `BranchDelayPCC s` returns both the address and the capability that is jumped to. `BranchTo s` and `BranchToPCC s` are ghost state that is used internally in the specification of branch instructions.
- `Mem s a` returns the memory contents at address `a`. Here, `a` is a 35-bit address that corresponds to a capability-sized and -aligned region of memory. The returned value is either `RawMemValue w` with `w` a 256-bit word, or `CapMemValue cap` with `cap` a capability (see Section 2.3.5).
- `ExceptionSignalled s` returns whether a hardware exception has been raised in the current execution step. `IsUnpredictable s` returns whether the state is unpredictable (see Section 2.3.4).

The parts of the machine state that are not necessary to define our security properties are still relevant to our proofs, as they can affect the behaviour of execution steps. These parts include exception registers, special arithmetic registers such as `HI` and `LO`, and most of the machine configuration. It also includes the state of programmable interrupt controllers (PICs), JTAG UART components, and the translation lookaside buffer (TLB). Finally, it includes fields that makes debugging the CHERI-MIPS specification easier, for example logging which addresses have been read from or written to. This debugging state is not part of the architectural state.

2.4.6 State monad

Recall that L3 is an imperative language, which means that there is a global state that functions can read and update. Isabelle, on the other hand, is a pure language, which means that functions cannot have side-effects such as accessing a global state. By adding the global state as an input parameter and as an extra output, one can transform an imperative function to a pure function. For example, an L3 function of type `'a ⇒ 'b` is exported to an Isabelle function of type `'a ⇒ State ⇒ 'b × State`, where `State` is the machine state described in Section 2.4.5.

When exporting a sequential composition of functions, the output state of the first function is used as the input for the second function. To keep this readable, the export from L3 to Isabelle uses a *state monad* of type `'s ⇒ 'a × 's`. Typically, `'s = State`, but the monad can be used with other types of state to support L3's mutable variables. For example, an L3 statement that uses mutable variables of types $b_1, \ldots, b_n$ is exported to an Isabelle/HOL function that uses a state monad with `'s =` $b_1 \times \ldots \times b_n \times$ `State`.

Below we describe the functions that can be used to manipulate monadic values.

`Return v`
: Transforms a value of type `'a` to a monadic value of type `'s ⇒ 'a × 's`. It simply returns `v` and the state that it receives as input, as can be seen from its formal definition: `Return v s ≡ (v, s)`.

`ReadState f`
: Instead of returning a fixed value, like `Return v`, this returns a value that depends on the state that it receives as input. More precisely, it transforms a function of type `'s ⇒ 'a` to a monadic value of type `'s ⇒ 'a × 's`. Its formal definition is: `ReadState f s ≡ (f s, s)`.

`UpdateState f`
: Changes the state and returns unit. More precisely, it transforms a function of type `'s ⇒ 's` to a monadic value of type `'s ⇒ Unit × 's`. Its formal definition is: `UpdateState f s ≡ ((), f s)`.

`Bind m n`

Combines two monadic values by sequentially composing them. Here, `m` has type `'s ⇒ 'a × 's`, and `n` has type `'a ⇒ 's ⇒ 'b × 's`. Given `s` as input, `Bind m n` first computes `m s`, which results in a value `a` and a new state `s'`. It then computes `n a s'` and returns its output. Its formal definition is as follows:

```
Bind m n s ≡ let (a, s') = m s in n a s'.
```

We use *do notation* to keep nested binds readable. For example, we write `Bind (Foo x) (λv.  Bind (Bar v) (λw.  Baz v w))` as:

```
do {
  v ← Foo x;
  w ← Bar v;
  Baz v w
}
```

`ExtendState v m`

Here, `m` is a monadic value of type `'b × 's ⇒ 'a × 'b × 's`, and `v` is a value of type `'b`. `ExtendState` changes `m` to a monadic value of type `'s ⇒ 'a × 's` by providing `v` as an extra input and by forgetting the value of type `'b` that `m` produces. More precisely, `ExtendState` receives a state `s` of type `'s` as input, combines this with `v` to form an input of `m`, and computes `m (v, s)`. This results in a value `a` and a new state `s'` of type `'b × 's`. It projects `s'` to its second component to obtain a state of type `'s`, and returns this together with the value `a`. Its formal definition is given below:

```
ExtendState v m s ≡
  let (a, s') = m (v, s) in
  (a, Second s')
```

Note that the name `ExtendState` can be confusing, since it trims the state of `m`. The name probably stems from the export of L3 to Isabelle: if one has to provide a value of type `'s ⇒ 'a × 's`, one can use `ExtendState v _` to introduce a mutable variable initialised to `v`, which means one now has to provide a value with an extended state.

`TrimState m`

Here, `m` is a monadic value of type `'s ⇒ 'a × 's`. `TrimState` changes `m` to a monadic value of type `'b × 's ⇒ 'a × 'b × 's` by passing along the value of type `'b` unchanged. More precisely, `TrimState` receives a state `s` of type `'b × 's` as input, projects this to its second component to form an input of `m`, and computes `m (Second s)`. This results in a value `a` and a new state `s'` of type `'s`. It combines `s'` with the first component of the original state `s` to form a state of type `'b × 's`, and returns this together with `a`. Its formal definition is given below:

```
TrimState m s ≡
  let (a, s') = m (Second s) in
  (a, First s, s')
```

Note that the name `TrimState` can be confusing, since it extends the state of `m`. The name probably stems from the export of L3 to Isabelle in a similar way as the name of `ExtendState`.

`ForEachLoop l m`

With `l` a list of type `'a List`, and `m` a monadic value of type `'a ⇒ 's ⇒ Unit × 's`, this applies `m` to each value in `l`, and returns its sequential composition. It has an inductive definition. If `l` is empty, it is defined by: `ForEachLoop [] m ≡ Return ()`. Otherwise, its definition is as follows, where `h # t` is the list with head `h` and tail `t`:

```
ForEachLoop (h # t) m ≡
  do {
    m h;
    ForEachLoop t m
  }
```

`ForLoop i j m`

With `m` a monadic value of type `Nat ⇒ 's ⇒ Unit × 's`, and `i` and `j` natural numbers, this applies `m` to each value in `i`, ..., `j`, and returns its sequential composition. The sequence decreases if `i > j`. Its formal definition below terminates because the absolute value of `i - j` decreases at each recursion.

```
ForLoop i j m ≡
  if i = j
  then m i
  else let i' = if i < j then i + 1 else i - 1 in
       do {
         m i;
         ForLoop i' j m
       }
```

2.4.7 Example exports

The Isabelle/HOL export of the L3 specification is too large to include here, so instead we just give an impression by including a few translated definitions. The translation of the CAPR state component (see Figure 2.15 on the following page) illustrates that its getter and setter functions are translated separately. The translation of the DADD instruction (see Figure 2.16 on the next page) shows that the Isabelle/HOL export is more verbose, as it uses several imperative steps to calculate the addition of two sign-extended registers. The translation of the `LoadCap` auxiliary function (see Figure 2.17 on page 78) is more difficult to read than its L3 source: deeply nested `Binds` obfuscate the control flow,

mutable variables are hard to distinguish, and its size increased from 42 to 84 lines. The translation of the `SignalException` auxiliary function (see Appendix A.2 on page 164) is even less readable, partly caused by its increase in size from 67 to 227 lines. The translated definitions do not need to be understood in detail to follow the contribution of this thesis; we include them here to show what kind of definitions our proof tactics need to handle.

```
ReadGPCapReg n ≡
  if n = 0
  then Return NullCap
  else do {
         reg ← ReadState _GPCapReg;
         Return (reg n)
       }
```

```
WriteGPCapReg (value, n) ≡
  do {
    reg ← ReadState _GPCapReg;
    reg' ← Return (reg(n := value));
    UpdateState (λs. s(_GPCapReg := reg'))
  }
```

Figure 2.15: The Isabelle export of the `CAPR` state component (see Figure 2.4 on page 56 for the L3 source). Note that we renamed the component to `GPCapReg` for readability. When exporting to Isabelle, the getter and setter function of state components are exported separately, with respectively the prefix `Read` and `Write`.

```
ExecuteDADD (rs, rt, rd) ≡
  do {
    v ← ReadGPR rs;
    temp ← do {
        v ← Return (SignedCast v);
        v0 ← ReadGPR rt;
        v0 ← Return (SignedCast v0);
        Return (v + v0)
      };
    if Bit temp 64 ≠ Bit temp 63
    then SignalException Overflow
    else WriteGPR (ExtractWord 63 0 temp, rd)
  }
```

Figure 2.16: The Isabelle export of the DADD instruction (see Figure 2.6 on page 58 for the L3 source).

```
LoadCap (vAddr, link) ≡
  do {
    (pAddr, cca, a_, l) ← AddressTranslation (vAddr, Load);
    b ← ReadExceptionSignalled;
    if b
    then do {
           v ← NextUnknown "capability";
           Return (Undefined v)
         }
    else let a = ExtractWord 39 5 pAddr in
         do {
           v ← ReadState JTAG_UART;
           _1 ← do {
               b ← do {
                   v ← do {
                       v ← Return (BaseAddress v);
                       Return (ExtractWord 36 2 v)
                     };
                   Return (a = v)
                 };
               if b
               then _RaiseL3Exception (Unpredictable "Capability load attempted on UART")
               else do {
                      v ← ReadState TotalCore;
                      j ← Return (v - 1);
                      ForLoop 0 j
                       (λcore. do {
                             v ← ReadState BaseAddressPIC;
                             base ← Return (v (IntToWord (NatToInt core)));
                             if ExtractWord 36 2 base ≤ a
                                and a < ExtractWord 36 2 (base + 1072)
                             then _RaiseL3Exception
                                    (Unpredictable "Capability load attempted on PIC")
                             else Return ()
                           })
                    }
             };
           _4 ←
             if link
             then do {
                    _2 ← if cca = 2
                         then _RaiseL3Exception (Unpredictable "load linked on uncached address")
                         else Return ();
                    _3 ← WriteLoadLinkFlag (Some True);
                    x ← ReadCP0;
                    WriteCP0 (x(_LoadLinkAddress := IntToWord (WordToInt pAddr)))
                  }
             else WriteLoadLinkFlag None;
```

(Continued on the next page)

Figure 2.17: The Isabelle export of the LoadCap auxiliary function (see Figure 2.9 on page 64 for the L3 source). This is page 1 of 2 of the figure.

```
49          v ← ReadCap a;
50          ExtendState v (do {
51                              _5 ← if l
52                                   then do {
53                                          v ← ReadState First;
54                                          v ← do {
55                                              v ← Return (v, False);
56                                              Return ((First v)(Tag := Second v))
57                                            };
58                                          UpdateState (λs. (v, Second s))
59                                        }
60                                   else Return ();
61                              v ← ReadState (λs. MemAccessStats (Second s));
62                              _6 ← do {
63                                  v ← do {
64                                      v ← do {
65                                          v0 ← ReadState (λs. MemAccessStats (Second s));
66                                          v0 ← do {
67                                              v ← Return (BytesRead v0);
68                                              Return (v + 32)
69                                            };
70                                          Return (v, v0)
71                                        };
72                                      Return ((First v)(BytesRead := Second v))
73                                    };
74                                  UpdateState (λs. (First s, (Second s)(MemAccessStats := v)))
75                                };
76                              v ← ReadState First;
77                              _7 ← do {
78                                  v ← Return (pAddr, v);
79                                  TrimState (WatchForCapLoad v)
80                                };
81                              ReadState First
82                            })
83        }
84  }
```

Figure 2.17: This is page 2 of 2 of the figure.

2.5 Augmenting the specification

The L3 specification does not define some aspects that are necessary to state and prove our security properties. To solve this, we augment the specification with some of our own auxiliary functions, defined in Isabelle/HOL. We formalise when a machine state is a valid starting state for CHERI-MIPS (see Section 2.5.1), which is left implicit in CHERI's documentation. We then define an auxiliary function that returns whether a hardware exception occurred during an execution step (see Section 2.5.2), and auxiliary functions that return memory contents as bytes or as a capability (see Section 2.5.3), which is defined in the L3 specification, but in a less convenient way. Finally, we formalise unpredictable behaviour (see Section 2.5.4), which is only described in prose [164, §9.1], and not captured in the L3 specification.

2.5.1 Valid states

The CHERI-MIPS specification has implicit assumptions on which states can be used as the starting state of an execution step. We call states that satisfy these requirements *valid* states. Invalid states can still be used as input of the function that computes an execution step, but then the resulting state will be meaningless. Two of the assumptions provide insight into the semantics of CHERI-MIPS, namely the fact that CHERI-MIPS only supports big-endian mode, and that it does not support reversing the endianness for user mode code. The other assumptions are technical: they are requirements about how the semantics of CHERI-MIPS is specified, not about the semantics itself. These assumptions are left implicit in the specification, but we need to make them explicit to be able to state security properties.

We capture the requirements with the function `StateIsValid`. We give its formal definition below and then discuss its individual clauses.

```
1  StateIsValid s ≡
2    BigEndian s
3    and not ReverseEndian s
4    and not ExceptionSignalled s
5    and not IsUnpredictable s
6    and BranchTo s = None
7    and BranchToPCC s = None
8    and BranchDelay s = None or BranchDelayPCC s = None
```

Lines 2, 3. A state `s` is only valid if `BigEndian s = True` and `ReverseEndian s = False`, as explained above.

Line 4. The L3 specification uses `ExceptionSignalled` to check whether a previously called auxiliary function raised an exception. If `ExceptionSignalled s = True` for a starting state `s`, the specification wrongfully assumes that an auxiliary function raised

an exception, resulting in an incorrect next state. Therefore, a state `s` is only valid if `ExceptionSignalled s = False`.

Line 5. The L3 specification indicates unpredictable behaviour by setting `L3Exception s` to a value `Unpredictable m`, which consequently gives `IsUnpredictable s` the value `True`. If `IsUnpredictable s` is already true for a starting state `s`, `IsUnpredictable s'` will be true for the next state `s'` even if no unpredictable behaviour occurred in that execution step. Therefore, the state `s` is only valid if `IsUnpredictable s = False`. Because an unpredictable state cannot be used as the starting state of the next step, execution traces get "stuck" whenever unpredictable behaviour occurs. This is caused by the fact that the L3 specification only specifies whether unpredictable behaviour occurs, but not what its behaviour is. We fix this in Section 2.5.4 by defining the semantics of unpredictable behaviour ourselves, allowing an execution to continue after an unpredictable step.

Lines 6, 7. During an execution step, branches update either `BranchTo` or `BranchToPCC` with a value other than `None`. At the end of the execution step, the values of `BranchTo` and `BranchToPCC` are copied to respectively `BranchDelay` and `BranchDelayPCC` regardless of whether a branch was executed, and `BranchTo` or `BranchToPCC` are cleared. If `BranchTo s` or `BranchToPCC s` already have a value other than `None` for a starting state `s`, the next state will contain the effects of a branch that did not actually take place. Therefore, a state `s` is only valid if `BranchTo s` and `BranchToPCC s` are both `None`.

Line 8. If `BranchDelay s` has a value `Some vAddr` for a starting state `s`, then `vAddr` will be copied to the PC at the end of the execution step. Similarly, if `BranchDelayPCC s` has a value `Some (vAddr', cap)`, then `vAddr'` and `cap` will be copied to respectively the PC and the PCC. This cannot both happen at the same time, so a state `s` is only valid if either `BranchDelay s = None` or `BranchDelayPCC s = None`.

With the semantics for unpredictable behaviour that we define in Section 2.5.4, we have that `StateIsValid` is an invariant of the semantics of CHERI-MIPS: if `s` is valid, then the next state `s'` is also valid. We prove this as part of Theorem 3.17 on page 111.

2.5.2 Hardware exceptions

In the context of the security guarantees that CHERI-MIPS offers, an execution step that raises a hardware exception is fundamentally different from one that does not: hardware exceptions are protection domain transitions, like capability invocation, which can change the capabilities that are available to executing code (see Section 1.3.6). Despite their importance, there are situations where the architecturally visible state does not clearly indicate whether an exception occurred. To see this, consider the following (contrived) example.

Example 2.1. Let `s` be a state, and suppose that at an earlier point an arithmetic overflow exception has been raised that is still being handled at `s`. This means that `ExceptionLevel s = True`, `ExceptionCode s` indicates that an overflow exception occurred, `PCC s = KernelCodeCap s`, and `BadInstr s` contains the opcode of the instruction that caused it. Furthermore, assume that the execution step before `s` branched to the exception handler of overflow exceptions, which means that `s` is in a branch delay slot. Finally, assume that the execution step from `s` to `s'` executes the same opcode that caused the original overflow exception.

The last execution step might also raise an overflow exception, but we cannot easily detect this: a second overflow exception would not change the values of the `PCC`, `ExceptionLevel`, `ExceptionCode`, and `BadInstr`, because of the original overflow exception. Furthermore, `PC s'` points to the exception handler regardless of the occurrence of a second overflow exception, because of the pending branch. The only way to tell whether an exception occurred is by inspecting the semantics of the arithmetic instruction, and reasoning whether it should overflow in state `s`.

To ensure that our security properties cover corner cases like the example above, we inspect the semantics of all instructions to determine when exceptions occur. Fortunately, the ghost state `ExceptionSignalled` has a very similar purpose: it is used in the L3 specification to check whether a previously called auxiliary function raised an exception. It would have been convenient if `ExceptionSignalled s'` indicated whether an exception occurred in the execution step from `s` to `s'`. Unfortunately, `Next`, the function that defines an entire execution step, clears `ExceptionSignalled` at the end. We circumvent this by defining `_Next`, which is a copy of `Next` except that it does not clear `ExceptionSignalled`. We prove the following lemma in Isabelle/HOL to show that we defined `_Next` correctly.

Lemma 2.2. The behaviour of `Next` equals the behaviour of `_Next` followed by clearing `ExceptionSignalled`. This is expressed formally as follows:

```
1 Next = do {
2   _Next;
3   v ← ReadState ProcessorState;
4   UpdateState (λs. s(ProcessorState := v(_ExceptionSignalled := False)))
5 }
```

2.5.3 Memory accesses

We define some auxiliary functions that make it easier to refer to memory accesses in our security properties.

Conceptually, the memory of CHERI-MIPS is a large array of bytes where each capability-sized and -aligned region has an associated tag. The L3 specification, however, models the memory as an array of items that are either a word or a capability:

`Mem s a` returns either `RawMemValue w` with `w` a 256-bit word, or `CapMemValue cap` with `cap` a capability, as we saw in Section 2.4.5. This speeds up the simulation of the specification, because memory contents that are repeatedly used as capabilities do not need to be repeatedly converted. For our security properties, on the other hand, the distinction between `RawMemValue` and `CapMemValue` is not meaningful. We therefore define the auxiliary functions `MemCap` and `MemByte` that return memory contents respectively as capabilities or as bytes.

The function `MemCap` is the simpler of the two. It takes a 35-bit address `a` as parameter, and returns the capability at that address, converting words to capabilities with `BitsToCap` if necessary. Its formal definition is given in Figure 2.18 on the next page. The function `MemByte` takes a 40-bit address `a` as parameter, and returns the byte at that address. First, it retrieves the memory contents of the capability-sized and -aligned region that `a` lies in, converting capabilities to words with `CapToBits` if necessary. It then extracts the correct byte, accounting for the fact that `Mem` stores bytes in little-endian, while CHERI-MIPS is a big-endian architecture. Its formal definition is given in Figure 2.19 on the following page.

We also define a function that translates addresses. The function `AddressTranslation`, which is exported from L3, can affect the machine state, for example by raising hardware exceptions or performing unpredictable operations. The translated address has a garbage value in those cases. We define an auxiliary function `TranslateAddr` that does not change the state, and that returns `Some pAddr` if the address can be translated, and `None` otherwise. It is defined by computing `AddressTranslation`, checking the resulting state for exceptions and unpredictable behaviour, and then discarding the resulting state. Its formal definition is given in Figure 2.20 on the next page.

2.5.4 Unpredictable behaviour

As explained in Section 2.3.4, the L3 specification defines when unpredictable behaviour occurs, but not what its semantics is. Because we cannot state our security properties without this, we formalise the prose semantics of unpredictable behaviour ourselves.

The CHERI-MIPS documentation requires the following: “For the CHERI mechanism to be secure, we require that programs whose behaviour is ‘unpredictable’ according to the MIPS ISA do not modify memory or capability registers in a way that allows the capability mechanism to be bypassed” [164, §9.1]. Unfortunately, the documentation does not explain what “bypassing the capability mechanism” precisely means.

The documentation continues with a concrete suggestion: “One easy way to achieve [that the capability mechanism cannot be bypassed by unpredictable behaviour] is that the ‘unpredictable’ case requires that neither memory nor capability registers are modified” [164, §9.1]. Unfortunately, this requirement is not strong enough to ensure that

```
1 MemCap s a ≡
2   case Mem s a
3   of CapMemValue cap ⇒ cap
4      RawMemValue x ⇒ BitsToCap x
```

Figure 2.18: The definition of `MemCap`, which returns the memory contents at the (35-bit) address `a` in state `s` as a capability. It uses `Mem` to retrieve the memory contents (Line 2). If the stored value is a capability, it returns that (Line 3), and otherwise it converts the stored word to an (invalid) capability and returns the result (Line 4).

```
1 MemByte s a ≡
2   let upper = Slice 5 a in
3   let word256 = case Mem s upper
4                 of CapMemValue cap ⇒ CapToBits cap
5                    RawMemValue x ⇒ x in
6   let lower = a AND Mask 5 in
7   let bigEndian = lower XOR Mask 3 in
8   ExtractByte (WordToNat bigEndian) word256
```

Figure 2.19: The definition of `MemByte`, which returns the memory contents at the (40-bit) address `a` in state `s` as a byte. It first selects the upper 35 bits of `a`, which specifies a capability-sized and -aligned region of memory (Line 2). It then retrieves the memory contents of that region (Line 3), converting capabilities to 256-bit words (Line 4). It selects the lower 5 bits of `a`, which specifies a byte within a capability-sized region (Line 6). It flips the lowest three bits to account for CHERI-MIPS's big-endian mode (Line 7). Finally, it extracts the correct byte from the 256-bit word and returns that (Line 4).

```
1 TranslateAddr (vAddr, t) s ≡
2   let ((pAddr, x), s') = AddressTranslation (vAddr, t) s in
3   if ExceptionSignalled s' or IsUnpredictable s'
4   then None
5   else Some pAddr
```

Figure 2.20: The definition of `TranslateAddr`, which returns the translation of the virtual address `vAddr`, for the access type `t` (which can be `Load` or `Store`) in state `s`. It uses `AddressTranslation` to obtain a physical address `pAddr` and a resulting state `s'` (Line 2). It checks whether the translation caused an exception or unpredictable behaviour (Line 3). If so, it returns `None`, and otherwise `Some pAddr` (Line 4–5).

our security properties hold. We argue that this is a problem of the semantics of unpredictable behaviour, not of our security properties. First, the requirement does not forbid changes to the TLB, which can be used to bypass the capability mechanism, as explained in the documentation [164, §9.2]. Second, the requirement does not forbid changes to `BranchDelayPCC`, because it is not a capability register, but changing it to a capability with permissions to the entire address space bypasses the capability mechanism. Finally, the requirement does not forbid producing invalid states (we defined validity in Section 2.5.1). We therefore propose the extra requirements that unpredictable behaviour cannot produce invalid states, and that is cannot change the `BranchDelayPCC` or any address translations.

We formalise the semantics of unpredictable behaviour as follows. Suppose the execution step starting at state `s` has unpredictable behaviour. Then `UnpredictableNext s` returns the set of states that are possible results of that unpredictable behaviour. Note that `UnpredictableNext` can take any state `s` as input, regardless of whether the execution step starting at `s` actually has unpredictable behaviour or not. Its formal definition is shown in Figure 2.21. The semantics of a (predictable or unpredictable) execution step is then defined as follows. Let `s' = Second (Next s)`. If `IsUnpredictable s'`, the resulting states are given by `UnpredictableNext s`, otherwise the resulting state is `s'`.

```
1  s' ∈ UnpredictableNext s ≡
2    for all a. MemCap s' a = MemCap s a
3    and PCC s' = PCC s
4    and for all cd. GPCapReg s' cd = GPCapReg s cd
5    and for all cd. SpecialCapReg s' cd = SpecialCapReg s cd
6    and BranchDelayPCC s' = BranchDelayPCC s
7    and for all vAddr. TranslateAddr vAddr s' = TranslateAddr vAddr s
8    and StateIsValid s'
```

Figure 2.21: The definition of `UnpredictableNext`, which returns the set of states that are possible results of an unpredictable execution step that starts at `s`. A state `s'` is contained in `UnpredictableNext s` if and only if the following conditions hold: the memory contents (including tags) are the same in `s` and `s'` (Line 2); the PCC, the general capability registers, and the special capability are the same in `s` and `s'` (Lines 3–5); the branch delay PCC is the same in `s` and `s'` (Line 6); address translation is the same in `s` and `s'` (Line 7); and the state `s'` is valid (Line 8).

Chapter 3

An abstraction of the CHERI-MIPS architecture

Here we define an abstraction that enables reasoning about malicious or compromised code that can execute arbitrary instructions. In principle, one can already reason about such code using the information provided by the CHERI-MIPS architecture. However, this is not viable in practice: the architecture contains around 200 instructions, described in 7k lines of specification, and every instruction needs to be considered. At this abstraction level, manual reasoning is too tedious and error-prone to be feasible, and automated reasoning, such as symbolic execution or model checking, quickly suffers from a combinatorial explosion. To solve this, we create a new abstraction level that describes memory accesses, capability manipulations, and hardware exceptions, and that abstracts away from all other instruction behaviour.

Before we define our abstraction, we give semantics to *malformed* capabilities by overestimating their authority (see Section 3.1). We then define an order over capabilities that captures when the authority of a capability is contained in the authority of another capability (see Section 3.2). This allows our abstraction to unify all instructions that *restrict* capabilities. Finally, we define *capability locations*, which allows us to treat capabilities in registers and memory in a uniform way (see Section 3.3).

Our abstraction uses *abstract actions* that capture four kinds of memory accesses (loading and storing data, and loading and storing capabilities), four kinds of capability manipulations (restricting, sealing, unsealing, and invoking it), and hardware exceptions (see Section 3.4). We then define the semantics of the abstraction by specifying under what conditions the actions can be performed, and what effects they have (see Section 3.5). Finally, we map all CHERI-MIPS instructions to abstract actions, capturing their intention, and we prove that our abstraction can simulate CHERI-MIPS through this mapping (see Section 3.6). This means that any CHERI-MIPS execution satisfies the properties of the abstraction, even executions of malicious or compromised code.

3.1 Malformed capabilities

Recall that a capability `cap` has authority to the memory region that starts at `Base cap` and ends at `Base cap + Length cap - 1`. This region is only well-defined if the end does not wrap, in other words, if `Base cap` $\leq$ `Base cap + Length cap - 1`. If it does wrap, we call the capability *malformed*, as opposed to *well-formed*.

Capability registers are initialised with well-formed capabilities on system reset, and CHERI-MIPS instructions never derive malformed capabilities from well-formed ones [164, §9.3]. However, memory is not cleared on system reset, which means it might contain malformed capabilities. To ensure that software does not encounter these, the CHERI-MIPS documentation requires that system software clears memory before using it or passing it to other code. CHERI-MIPS instructions can therefore disregard malformed capabilities, which means their bounds checks can be simplified. There are other reasons behind the requirement, for example, uncleared memory can also contain highly privileged, well-formed capabilities and sensitive data from prior boots.

We could make the same assumption in our security properties. However, the aim of our security properties is to capture the guarantees of the architecture regardless of the correctness of (system) software that runs on top. To achieve this, we give semantics to malformed capabilities: we define that malformed capabilities have authority over the entire address range. The property that bounds checks only succeed if the capability has enough authority then trivially holds for malformed capabilities. The other direction does not hold: bounds checks can reject malformed capabilities even though they have enough authority, but this does not pose a problem for our security properties.

We formalise the memory region of a capability as follows. First, we define the region starting from `b` with length `l`, both of type `'a Word`, resulting in a set of `'a Word`. Line 2 below checks whether `b + l - 1` wraps: it converts both `b` and `l` to natural numbers, and checks whether their addition is larger than $2^{|\texttt{'a}|}$. If so, it returns the set of all `'a Words` (Line 3). Otherwise, it returns the set `b + 0, ..., b + (l - 1)`, which equals the empty set if `l = 0` (Line 4). Note that `{x. P x}` is Isabelle's syntax for the set of all `x` for which `P x` holds.

```
1 Region b l ≡
2   if WordToNat b + WordToNat l > 2^Size b
3   then {addr. True}
4   else {b + offset |offset. offset < l}
```

The memory region of a (well-formed or malformed) capability is then defined as:

```
RegionOfCap cap ≡ Region (Base cap) (Length cap).
```

3.2 Capability order

We define an order, $\leq$, over capabilities, capturing when the authority of a capability is contained in the authority of another capability. This order is based on the following observations.

- Invalid capabilities have no authority.
- The authorities of (valid) sealed and unsealed capabilities are incomparable. For example, unsealed capabilities can authorise memory accesses and sealed capabilities can be invoked (with the right permissions), but not vice versa.
- The authority of valid, unsealed capabilities can be decreased by shrinking their bounds and removing permissions. Changing the address of an unsealed capability does not change its authority.
- The above does not hold for valid, sealed capabilities: a sealed capability with the PermitExecute permission cannot be invoked as a data capability, but removing the permission might make it suitable. Furthermore, the address and the object type of a sealed capability determine how it can be unsealed, so changing either field changes its authority. We conservatively regard a change to the bounds of a valid, sealed capability also as a change of authority.
- We do not know how the reserved bits of capabilities will be interpreted, so we regard a change in these bits as an incomparable change of authority.

These observations lead to the following formal definition.

Definition 3.1 (Order over capabilities)**.** We say `cap ≤ cap'` if either `cap` is invalid (Line 2 below), or `cap` and `cap'` are equal (Line 3), or both capabilities are valid and unsealed (Lines 4–5) and: the memory region of `cap` is contained in the memory region of `cap'` (Line 6), the (system and user) permissions of `cap` are less then or equal to those of `cap'` (Lines 7–8), their object types agree (Line 9), and their reserved bits agree (Line 10). Note that Lines 4–10 do not constrain the addresses of `cap` and `cap'`.

```
1  cap ≤ cap' ≡
2    not Tag cap
3    or cap = cap'
4    or Tag cap and Tag cap'
5       and not IsSealed cap and not IsSealed cap'
6       and RegionOfCap cap ⊆ RegionOfCap cap'
7       and Perms cap ≤bitwise Perms cap'
8       and UserPerms cap ≤bitwise UserPerms cap'
9       and ObjectType cap = ObjectType cap'
10      and Reserved cap = Reserved cap'
```

This order is reflexive and transitive (a preorder). It is not antisymmetric: if `cap` and `cap'` are valid and unsealed, and differ only by their addresses, we have `cap ≤ cap'`,

`cap'` $\leq$ `cap`, but `cap` $\neq$ `cap'`. The preorder is also not total: if `cap` and `cap'` are respectively the sealed and unsealed version of the same capability, then `cap` $\not\leq$ `cap'` and `cap'` $\not\leq$ `cap`.

3.3 Capability locations

The machine state contains capabilities in memory, in the branch delay PCC, and in various registers, such as the PCC, the general purpose, and the special capability registers. We introduce *capability locations* to be able to handle these in a uniform way. This allows our abstraction to express when the precise location of a capability is relevant and when it is not.

First, we define the tagged union type `CapRegister`: the constructor `RegPCC` refers to the PCC, the constructor `RegGeneral` with parameter `i` refers to the `i`-th general purpose capability register, and the constructor `RegSpecial` with parameter `i` refers to the `i`-th special capability register. For convenience, we also define a constructor `RegBranchDelayPCC` that refers to the branch delay PCC, although strictly speaking this is not a register. Finally, we define a constructor `RegBranchToPCC` that refers to the ghost state `BranchToPCC`. We include this ghost state because our proof tactics refer to it. We then define the function `CapReg` that returns the capability in state `s` at the capability register `r`. It converts `None` to `NullCap` when necessary, but otherwise its definition is straightforward:

```
1  CapReg s r ≡
2    case r
3    of RegPCC ⇒ PCC s
4       RegBranchDelayPCC ⇒ case BranchDelayPCC s
5                           of None ⇒ NullCap
6                              Some (x, cap) ⇒ cap
7       RegBranchToPCC ⇒ case BranchToPCC s
8                        of None ⇒ NullCap
9                           Some (x, cap) ⇒ cap
10      RegGeneral i ⇒ GPCapReg s i
11      RegSpecial i ⇒ SpecialCapReg s i
```

Second, we define the tagged union type `CapLocation`, which can also refer to capabilities in memory: the constructor `LocReg r` refers to the capability in register `r`, and `LocMem a` refers to the capability at the (35-bit) address `a` in memory. We then define the function `Cap` that returns the capability in state `s` at the capability location `loc`:

```
Cap s loc ≡
  case loc
  of LocReg r ⇒ CapReg s r
     LocMem a ⇒ MemCap s a
```

3.4 Abstract actions

Our abstraction uses *abstract actions* to describe memory accesses, capability manipulations, and hardware exceptions. They contain information about the operation, such as memory footprints, source and destination registers, and the register of the capability that is used as authority.

We first consider the operations that cause a protection domain switch, namely capability invocation and hardware exceptions (see Section 1.3.6). Recall that capabilities are invoked in pairs that consist of a code and a data capability. When an instruction invokes a pair of capabilities, we retain the registers of these capabilities. When an instruction raises an exception, we only retain the fact that an exception occurred, but we abstract away from the instruction that caused it and the type of exception. We capture this with the tagged union type `DomainSwitchingAction`, which has the following constructors.

- `InvokeCapability` has parameters `r`, the register of the code capability, and `r'`, the register of the data capability that is invoked.
- `RaiseException` has no parameters.

We then consider the operations that preserve the protection domain. These are best explained through some examples, which all assume that no exception is being raised. The CAndPerm instruction copies a capability and removes permissions from the copy. We retain the source and destination register, and the fact that it restricts the copy according to the order we defined in Section 3.2, but we abstract away from how exactly it restricts the copy. For the CLoad instruction, we retain the fact that it loads data, the register of the capability that is used as authority, and the address and length of the access, but we abstract away from the destination register and the value that is loaded. Similarly, for the CStore instruction, we abstract away from the source register and the value that is stored. For instructions that load or store capabilities as opposed to data, we do retain respectively the destination and the source register, which allows us to determine the capability that is loaded or stored. We capture these operations, and sealing and unsealing capabilities, with the tagged union type `DomainPreservingAction`:

- `LoadDataAction` has parameters `auth`, the register of the capability that is used as authority, `a`, the physical address of the data, and `l`, the length of the data that is loaded. `StoreDataAction` is the analogue for stores.
- `LoadCapAction` has parameters `auth`, the register of the capability that is used as authority, `a`, the physical address of the capability that is loaded, and `r`, the destination register. `StoreCapAction` is the analogue for storing capabilities, except here `r` is the source register and `a` the physical address of the destination.
- `RestrictCapAction` has parameters `r`, the source register, and `r'`, the destination register where a restricted version of the source is copied to.

- `SealCapAction` has parameters `auth`, the register of the capability that is used as authority, `r`, the source register, and `r'`, the destination register where a sealed version of the source is copied to. `UnsealCapAction` is the analogue for unsealing capabilities.

Finally, we define the type `AbstractStep` that represents the abstraction of an entire execution step. It distinguishes steps that preserve the domain from those that switch domains. An execution step that preserves the domain might be abstracted to multiple actions. This is necessary to support, for example, the CJALR "jump and link capability register" instruction, which manipulates two capabilities, and the ClearLo and ClearHi instructions, which each manipulate up to sixteen capabilities. An execution step that preserves the domain might also be abstracted to no actions. This is necessary to support instructions that do not access memory nor manipulate capabilities, which is the majority of CHERI-MIPS's instructions. We capture all this in the tagged union type `AbstractStep`:

- `SwitchDomain` has parameter `action`, an action of type `DomainSwitchingAction`.
- `PreserveDomain` has parameter `actions`, a set of actions, each of type `DomainPreservingAction`.

In Section 3.6 we map CHERI-MIPS execution steps to abstract steps, but first we give semantics to our abstraction.

3.5 Abstract semantics

We define the semantics of our abstraction as a labelled transition system, where execution steps have a starting state `s`, a label of type `AbstractStep`, and a resulting state `s'`, with both `s` and `s'` concrete CHERI-MIPS machine states. The labels are interpreted as the intention of the execution step, following the principle of intentional use (see Section 1.3.1). This enables us to forbid the execution step `(s, l1, s')`, while allowing the step `(s, l2, s')` that has the same starting and resulting state, but a different intention. Our abstraction allows any execution step that satisfies the security properties that we define in this section.

The security properties have two design goals. First, they should be strong enough to enable reasoning about malicious or compromised code that can execute arbitrary instructions. In particular, we are interested in which memory locations such code can access and which protection domain transitions it can perform. We therefore define properties that describe under what conditions one can access memory (Section 3.5.1) or perform domain transitions (Section 3.5.3). To determine whether arbitrary code can satisfy these conditions, we need to reason about which capabilities it can access or construct. We

therefore define properties that describe the conditions and effects of capability manipulations (Section 3.5.2). This includes a property about address translation: changing address translations indirectly manipulates the authority of capabilities, since their bounds are virtual addresses.

The second design goal is that the properties are weak enough to allow all behaviour of CHERI-MIPS. This leads to some subtleties in our security properties. For example, the interaction between the capability system and virtual memory based protection depends on the type of the memory access. If one has a capability that can authorise storing another capability, but the corresponding TLB entry forbids it, then CHERI-MIPS raises an exception. However, if one has a capability that can authorise *loading* another capability, but the corresponding TLB entry forbids it, CHERI-MIPS still loads the capability, but clears the tag of the result.

We define the properties below. In the next chapter we show that they are indeed strong enough to reason about a concrete compartmentalisation scenario and in Chapter 5 we discuss our proof that the abstraction can indeed simulate CHERI-MIPS.

3.5.1 Memory accesses

Here we define properties about various kinds of memory accesses. We define separate properties for loading and storing capabilities, as opposed to loading and storing data, for two reasons. First, loading and storing capabilities requires additional permissions, namely PermitLoadCapability and PermitStoreCapability. This enables compartmentalisation scenarios where compartments can exchange data via a shared region of memory, but not capabilities. Second, accessing a capability is a capability manipulation. As discussed above, we need to describe the effects of capability manipulations to be able to reason about which memory locations future execution steps might access.

The property about loading data is given below. It requires that the capability that is used as authority is valid, is unsealed, has the PermitLoad permission, and has authority to the footprint of the access. The latter is determined by translating all virtual addresses that the capability has authority to, and checking whether those cover the footprint. Just translating the bounds of the capability does not suffice, because a contiguous region of virtual memory does not need to map to a contiguous region of physical memory.

Property 3.2 (Loading data)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that data is loaded, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `LoadDataAction auth a ln` is one of those actions (Line 5). Recall that `auth` is the register of the capability that is used as authority, `a` the physical address that is accessed, and `ln` the length of the access. The capability that is used as authority is then given by `CapReg s auth`. The property requires that it is valid (Line 6), is unsealed

(Line 7), and has the PermitLoad permission (Line 8). Furthermore, it requires that the length of the access is non-zero (Line 9). Finally, it requires that the footprint of the access is contained in the translation of the memory region that the capability has authority to (Line 10). The auxiliary function `TranslateAddresses` is similar to `TranslateAddr` except it translates a set of virtual addresses instead of a single address. It is formally defined in Appendix A.3.

```
1  LoadDataProp s label s' ≡
2    for all actions auth a ln.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and LoadDataAction auth a ln ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitLoad (CapReg s auth)
9         and ln ≠ 0
10        and Region a ln
11            ⊆ TranslateAddresses (RegionOfCap (CapReg s auth)) Load s
```

The property about storing data has similar requirements on the state `s`, except it requires the PermitStore permission. It also describes the effects on the resulting state `s'`. Here we account for the following design decision in the L3 specification: when storing to an address that is in use by a UART device or a PIC, the L3 specification updates their state but leaves the `Mem` field unchanged. UART devices and PICs are not represented in our abstraction, so we simply state that storing data might not have an effect on the memory, without specifying the circumstances in which that happens. If it does have an effect on the memory, it must clear the tag of the corresponding capability-sized and -aligned region.

Property 3.3 (Storing data)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that data is stored, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `StoreDataAction auth a ln` is one of those actions (Line 5). The capability that is used as authority is given by `CapReg s auth`. The property requires that it is valid (Line 6), is unsealed (Line 7), and has the PermitStore permission (Line 8). Furthermore, it requires that the length of the access is non-zero (Line 9), and that the footprint of the access is contained in the translation of the memory region that the capability has authority to (Line 10). Finally, it requires that either the tag of the capability-sized and -aligned region is cleared (Line 12), or that the entire region, including tag, remains unchanged (Line 13). Here we use the auxiliary function `GetCapAddress` to convert a 40-bit address to the address of the capability-sized and -aligned region, which simply removes the 5 least significant bits: `GetCapAddress a ≡ Slice 5 a`.

```
1  StoreDataProp s label s' ≡
2    for all actions auth a ln.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and StoreDataAction auth a ln ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitStore (CapReg s auth)
9         and ln ≠ 0
10        and Region a ln
11            ⊆ TranslateAddresses (RegionOfCap (CapReg s auth)) Store s
12        and not Tag (MemCap s' (GetCapAddress a))
13            or MemCap s' (GetCapAddress a) = MemCap s (GetCapAddress a)
```

The property about executing instructions has similar requirements, except for the following: the capability that is used as authority is fixed, namely the PCC; it requires the PermitExecute permission; the address of the instruction is specified in virtual memory, so there is no need to translate the memory region of the capability; and there is no abstract action that corresponds to executing an instruction. Instead, we require the property to always hold, unless the execution step raises an exception.

Property 3.4 (Executing instructions)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 2) and that the label indicates that no exception is raised (Line 3). The property then requires that the PCC is valid (Line 4), is unsealed (Line 5), has the PermitExecute permission (Line 6), and that the address of the instruction lies within its bounds (Line 7).

```
1  ExecuteProp s label s' ≡
2    if StateIsValid s
3       and label ≠ SwitchDomain RaiseException
4    then Tag (PCC s)
5         and not IsSealed (PCC s)
6         and PermitExecute (PCC s)
7         and Base (PCC s) + PC s ∈ RegionOfCap (PCC s)
```

The property below describes the effects of loading a capability from address `a` to register `r`. This includes the corner case mentioned at the beginning of this section: if the capability that is used as authority allows loading a capability from `a`, but the corresponding TLB entry forbids it, the capability is still loaded into register `r`, but the tag of the resulting capability is cleared. Our property only aims to capture the guarantees of the capability system, and not those of virtual memory based protection, so it does not check the permissions in the TLB entry. To account for the possibility that the tag is cleared, it states that the capability in register `r` in the resulting state is less than or equal to the capability at address `a` in the original state.

Property 3.5 (Loading capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a capability is loaded, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `LoadCapAction auth a r` is one of those actions (Line 5). The capability that is used as authority is given by `CapReg s auth`. The property requires that it is valid (Line 6), is unsealed (Line 7), and has the PermitLoad and the PermitLoadCapability permissions (Lines 8–9). Furthermore, it requires that the footprint of the access is contained in the translation of the memory region that the capability has authority to (Line 10). To determine the footprint, we convert the 35-bit address `a` to a 40-bit address by appending 5 zeroes with the following auxiliary function: `ExtendCapAddress a ≡ WordCat a 0`, where `0` is a `5 Word`. The length of the footprint is 32 because we use 256-bit capabilities. Finally, the property requires that the capability in register `r` in the resulting state `s'` is less than or equal to the capability at address `a` in the original state `s` (Line 12).

```
1  LoadCapProp s label s' ≡
2    for all actions auth r a.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and LoadCapAction auth a r ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitLoad (CapReg s auth)
9         and PermitLoadCapability (CapReg s auth)
10        and Region (ExtendCapAddress a) 32
11              ⊆ TranslateAddresses (RegionOfCap (CapReg s auth)) Load s
12        and GPCapReg s' r ≤ MemCap s a
```

The property about storing capabilities is an analogue of the property about loading capabilities, except that the tag of the stored capability is always kept intact. This is caused by a different interaction with virtual memory based protection, as described in the beginning of this section: if the relevant TLB entry forbids storing capabilities, CHERI-MIPS raises an exception instead of clearing the tag of the stored capability. The property is defined below.

Property 3.6 (Storing capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a capability is stored, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `StoreCapAction auth r a` is one of those actions (Line 5). The capability that is used as authority is given by `CapReg s auth`. The property requires that it is valid (Line 6), is unsealed (Line 7), and has the PermitStore and the PermitStoreCapability permissions (Lines 8–9). Furthermore, it requires that the footprint of the access is contained in the translation of the memory region that the

capability has authority to (Line 10). Finally, the property requires that the capability at address a in the resulting state s' equals the capability in register r in the original state s (Line 12).

```
1  StoreCapProp s label s' ≡
2    for all actions auth r a.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and StoreCapAction auth r a ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitStore (CapReg s auth)
9         and PermitStoreCapability (CapReg s auth)
10        and Region (ExtendCapAddress a) 32
11            ⊆ TranslateAddresses (RegionOfCap (CapReg s auth)) Store s
12        and MemCap s' a = GPCapReg s r
```

To store a local capability one needs an additional permission, namely PermitStoreLocalCapability. The property below describes this.

Property 3.7 (Storing local capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a capability is stored, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `StoreCapAction auth r a` is one of those actions (Line 5). Furthermore, assume that the capability that is stored is valid (Line 6) and local (Line 7). Then, the capability that is used as authority must have the PermitStoreLocalCapability permission (Line 8).

```
1  StoreLocalCapProp s label s' ≡
2    for all actions auth r a.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and StoreCapAction auth r a ∈ actions
6       and Tag (GPCapReg s r)
7       and not IsGlobal (GPCapReg s r)
8    then PermitStoreLocalCapability (CapReg s auth)
```

Finally, we define a property that only allows the memory at address `a` to be changed if the label of the execution step contains a `StoreDataAction` or a `StoreCapAction` action with `a` in its footprint. It uses the auxiliary function `WrittenAddresses action` that returns the footprint that `action` stores to. The footprint of `StoreDataAction auth a ln` starts at `a` and has length `ln`, the footprint of `StoreCapAction auth r a` starts at `a` (extended to a 40-bit address) and has length 32, and other actions do not store anything. Its formal definition is given below.

```
1 WrittenAddresses action ≡
2   case action
3   of StoreDataAction auth a ln ⇒ Region a ln
4      StoreCapAction auth r a ⇒ Region (ExtendCapAddress a) 32
5      _ ⇒ {}
```

Using this, we can now define the property. It only requires the value at address `a` to remain unchanged, and not the tag. Property 3.14 on page 101, among other things, states the conditions when a memory tag remains unchanged.

Property 3.8 (Memory invariant)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that the domain is preserved (Line 4), and none of the actions store to the address `a` (Lines 5–7). Then, the memory contents at `a` must stay unchanged (Line 8).

```
1 MemoryInvariant s label s' ≡
2   for all actions a.
3   if StateIsValid s
4      and label = PreserveDomain actions
5      and not exists action.
6              action ∈ actions
7              and a ∈ WrittenAddresses action
8   then MemByte s' a = MemByte s a
```

The invariant above is the counterpart of the properties about storing data and storing capabilities: if an execution step contains a `StoreDataAction` or a `StoreCapAction` action with `a` in its footprint, then one of the latter properties applies, and otherwise the invariant applies.

Ideally, we would also define a counterpart for the properties about loading data and loading capabilities, but, unfortunately, we cannot, for the following reason. One would typically define such a counterpart as an information flow property, along the lines of "if `a` is not in the footprint of a `LoadDataAction` or a `LoadCapAction` action, then the execution step may not depend on the memory contents at `a`". However, CHERI-MIPS is a non-deterministic specification, which means its architectural visible behaviour can differ between implementations, but also vary over time on the same implementation [94, §1.2.1]. A (compromised or adversarial) implementation could exploit this to leak information. For example, the result of 32-bit unsigned addition is unpredictable when used on 64-bit values (see Figure 2.8 on page 62). An implementation that returns 0 if the memory contents at `a` equals 0, and that otherwise returns a non-zero garbage value, conforms to the specification, but invalidates the information flow property.

3.5.2 Capability manipulations

We defined the properties for loading and storing capabilities above, because they are memory accesses, and we define the property about capability invocation in the next subsection, because that causes a domain transition. Here we define properties for the remaining capability manipulations. Some of them are centred around an abstract action, namely the properties about restricting, sealing, and unsealing capabilities. We also define properties about address translation and system register access, and an invariant that states under which conditions a capability remains unchanged.

The property about restricting capabilities states that the resulting capability is less than or equal to the original capability in the order we defined in Section 3.2. One does not need any permission to restrict capabilities.

Property 3.9 (Restricting capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a capability is restricted, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `RestrictCapAction r r'` is one of those actions (Line 5). The property then requires that the capability in the destination register `r'` in the resulting state `s'` is less than or equal to the capability in the source register `r` in the original state `s` (Line 6).

```
1 RestrictCapProp s label s' ≡
2   for all actions r r'.
3   if StateIsValid s
4      and label = PreserveDomain actions
5      and RestrictCapAction r r' ∈ actions
6   then CapReg s' r' ≤ CapReg s r
```

The property below describes sealing capabilities. Recall that capabilities are sealed with a 24-bit object type. The object type `t` is determined by the address of the capability that is used as authority. Since the address is 64-bit, only the 24 least significant bits are used. The object type must lie in the memory region of the capability that is used as authority. To check this, the object type is padded with zeroes to form a 64-bit value again. The property below also describes the effects of sealing a capability: the result equals the original capability, except its `IsSealed` field is `True` and its `ObjectType` field equals `t`.

Property 3.10 (Sealing capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a capability is sealed, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `SealCapAction auth r r'` is one of those actions (Line 5). Then define the 24-bit object type `t` as the unsigned cast of the address of the

capability that is used as authority (Line 6). The property requires that the capability that is used as authority is valid (Line 7), is unsealed (Line 8), and has the PermitSeal permission (Line 9). Furthermore, it requires that `t`, cast to a 64-bit word, is contained in the memory region that the capability has authority to (Line 10). It also requires that the capability that is being sealed is not already sealed (Line 11). Finally, it requires that the capability in the destination register `r'` in the resulting state `s'` equals the capability in the source register `r` in the original state `s`, except it is sealed and has object type `t` (Line 12).

```
1  SealCapProp s label s' ≡
2    for all actions auth r r'.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and SealCapAction auth r r' ∈ actions
6    then let t = UnsignedCast (Address (CapReg s auth)) in
7         Tag (CapReg s auth)
8         and not IsSealed (CapReg s auth)
9         and PermitSeal (CapReg s auth)
10        and UnsignedCast t ∈ RegionOfCap (CapReg s auth)
11        and not IsSealed (GPCapReg s r)
12        and GPCapReg s' r' = (GPCapReg s r)(IsSealed := True, ObjectType := t)
```

The property about unsealing capabilities is similar, except for the following corner case. If the capability that is used as authority does not have the IsGlobal permission, then that permission is stripped from the unsealed capability, even if the original capability does have that permission. Our property abstracts away from these details by stating that the resulting capability must be less than or equal to the unsealed version of the original capability.

Property 3.11 (Unsealing capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a capability is unsealed, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and `UnsealCapAction auth r r'` is one of those actions (Line 5). The property requires that the capability that is used as authority is valid (Line 6), is unsealed (Line 7), and has the PermitUnseal permission (Line 8). Furthermore, it requires that the object type of the original capability, cast to a 64-bit word, is contained in the memory region of the capability that is used as authority (Line 9). It also requires that the original capability is not already unsealed (Line 11). Finally, it requires that the capability in the destination register `r'` in the resulting state `s'` is less than or equal to the unsealed version of the original capability (Line 12).

```
1  UnsealCapProp s label s' ≡
2    for all actions auth r r'.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and UnsealCapAction auth r r' ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitUnseal (CapReg s auth)
9         and UnsignedCast (ObjectType (GPCapReg s r))
10            ∈ RegionOfCap (CapReg s auth)
11        and IsSealed (GPCapReg s r)
12        and GPCapReg s' r' ≤ (GPCapReg s r)(IsSealed := False, ObjectType := 0)
```

The property below describes accessing special capability registers (SCRs). The property is not tied to a specific abstract action, but applies to each domain-preserving action that refers to an SCR. The function `SpecialRegisterParameters action` captures this: it returns the set of SCRs that `action` uses. For example, if the source register `r` of the action `RestrictCapAction r r'` is an SCR, but its destination register `r'` is a general purpose capability register, then `SpecialRegisterParameters` returns the set `{r}`. The function is defined formally in Appendix A.3. If an SCR other than register 0 or 1 is accessed, the property below requires that the PCC has the PermitAccessSystemRegisters permission. SCR 0 and 1 are respectively the DDC and the TLSC, which are always accessible. The property does not require that the PCC is valid and unsealed, as this is already covered by Property 3.4 about executing instructions.

Property 3.12 (Special capability register access)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that a special capability register is accessed, namely the label equals `PreserveDomain actions` for some set of actions (Line 4), and one of the actions (Line 5) uses the special capability register `r` (Line 6). Furthermore, assume that `r` does not equal 0 or 1, which is respectively the DDC and TLSC (Line 7). The property then requires that the PCC has the PermitAccessSystemRegisters permission (Line 8).

```
1  SpecialRegisterProp s label s' ≡
2    for all actions action r.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and action ∈ actions
6       and r ∈ SpecialRegisterParameters action
7       and r ≠ 0 and r ≠ 1
8    then PermitAccessSystemRegisters (PCC s)
```

Changing address translations indirectly changes the authority of capabilities, because the virtual memory region they have authority to might now map to a different region of

physical memory. The property below states that address translations remain unchanged if the execution step does not raise an exception and the PCC does not have the PermitAccessSystemRegisters permission. The first condition is necessary because exceptions promote the processor mode to kernel mode, which affects address translation, as described in Section 2.1.2. The second condition is necessary because one can change TLB entries if the PCC has that permission.

Property 3.13 (Address translation)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3), that the label indicates no exception is raised (Line 4), and that the PCC does not have the PermitAccessSystemRegisters permission (Line 5). Then, for any address `a`, its translation is the same in `s'` as in `s` (Line 6).

```
1 AddressTranslationProp s label s' ≡
2   for all a.
3   if StateIsValid s
4      and label ≠ SwitchDomain RaiseException
5      and not PermitAccessSystemRegisters (PCC s)
6   then TranslateAddr a s' = TranslateAddr a s
```

Finally, we define a property that only allows a capability to be changed if the change is described by an abstract action. We define the function `ActionTargets` to make this precise: it takes a domain-preserving action as parameter and returns the set of capability locations (see Section 3.3) that the action manipulates. This set is empty for the action about loading data. For the other actions, this set consists of the destination register or the destination address. We formally define the function below.

```
1 ActionTargets action ≡
2   case action
3   of LoadDataAction auth a ln ⇒ {}
4      StoreDataAction auth a ln ⇒ {LocMem (GetCapAddress a)}
5      RestrictCapAction r r' ⇒ {LocReg r'}
6      LoadCapAction auth a r ⇒ {LocReg (RegGeneral r)}
7      StoreCapAction auth r a ⇒ {LocMem a}
8      SealCapAction auth r r' ⇒ {LocReg (RegGeneral r')}
9      UnsealCapAction auth r r' ⇒ {LocReg (RegGeneral r')}
```

Using this, we can now define the property. It assumes that there is no action with location `loc` as its target, and it requires that the capability at that location stays the same. It only applies to execution steps that preserve the domain.

Property 3.14 (Capability invariant)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3), that the label indicates that the domain is preserved (Line 4), and that there is no action that has the location `loc` as its target (Lines 5–7). Then the capability at `loc` in the resulting state `s'` is the same as the capability at `loc` in the original state `s` (Line 8).

```
1  CapabilityInvariant s label s' ≡
2    for all actions loc.
3    if StateIsValid s
4       and label = PreserveDomain actions
5       and not exists action.
6               action ∈ actions
7               and loc ∈ ActionTargets action
8    then Cap s' loc = Cap s loc
```

3.5.3 Domain transitions

Here we define properties for execution steps that switch domains, namely steps that invoke capabilities or raise an exception. As usual, the properties describe the requirements and effects of these actions, but because the invariants we defined above (see Properties 3.8 on page 97 and 3.14 on the preceding page) only hold for domain-preserving steps, the properties here also describe which parts of the state remain unchanged. In hindsight, we could have strengthened the invariants to also apply to domain-switching steps, which would make the properties here less verbose. The property below applies to capability invocation.

Property 3.15 (Invoking capabilities)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 3) and that the label indicates that the capabilities in registers `r` and `r'` are invoked (Line 4). It then defines `codeCap` and `dataCap` as the capabilities in respectively register `r` and `r'` (Lines 5–6). The property requires that both these capabilities are valid (Line 7), are sealed (Line 8), have the permission to be invoked (Line 9), that the code capability has the permission to execute (Line 10), but the data capability does not (Line 11), and that both capabilities have the same object type (Line 12). Furthermore, it requires that the offset of the code capability is copied to the PC (Line 13), the unsealed code capability is copied to the PCC (Line 14), any pending branches are cancelled (Lines 15–16), the unsealed data capability is copied to the IDC (Line 17), all normal registers, except the IDC (register 26), remain unchanged (Line 18), all special registers remain unchanged (Line 19), and the (entire) memory remains unchanged (Line 20).

```
1  InvokeCapProp s label s' ≡
2   for all r r'.
3   if StateIsValid s
4      and label = SwitchDomain (InvokeCapability r r')
5   then let codeCap = GPCapReg s r in
6        let dataCap = GPCapReg s r' in
7        Tag codeCap and Tag dataCap
8        and IsSealed codeCap and IsSealed dataCap
9        and PermitCCall codeCap and PermitCCall dataCap
10       and PermitExecute codeCap
11       and not PermitExecute dataCap
12       and ObjectType codeCap = ObjectType dataCap
13       and PC s' = Offset codeCap
14       and PCC s' = codeCap(IsSealed := False, ObjectType := 0)
15       and BranchDelay s' = None
16       and BranchDelayPCC s' = None
17       and InvokedDataCap s' = dataCap(IsSealed := False, ObjectType := 0)
18       and for all cb. if cb ≠ 26 then GPCapReg s' cb = GPCapReg s cb
19       and for all cb. SpecialCapReg s' cb = SpecialCapReg s cb
20       and for all a. Mem s' a = Mem s a
```

The property below describes hardware exceptions. Exception handlers in CHERI-MIPS are located at a fixed set of addresses, which we name `ExceptionPCs`. An exception copies the kernel code capability (KCC) to the PCC and jumps to one of these addresses. Unless another exception is being handled, the original PCC is saved to the EPCC. Similar to capability invocation, an exception cancels any pending branches.

Property 3.16 (Raising exceptions)**.** An execution step `(s, label, s')` satisfies this property if the following holds. Assume that `s` is a valid state (Line 2) and that the label indicates that an exception is raised (Line 3). The property requires that the exception flag is set (Line 4), that the address of the next instruction if one of a fixed set of exception entry addresses (Line 5), that the KCC is copied to the PCC (Line 6), and that all the normal capability registers remain unchanged (Line 7). Unless the exception flag was already set, the PCC must be copied to the EPCC (Line 8). All special capability registers, except register 31 (the EPCC), must remain unchanged (Line 11). Furthermore, the (entire) memory must remain unchanged (Line 12), and any pending branches must be cancelled (Lines 13–14).

```
ExceptionProp s label s' ≡
  if StateIsValid s
     and label = SwitchDomain RaiseException
  then ExceptionLevel s'
       and Base (PCC s') + PC s' ∈ ExceptionPCs
       and PCC s' = KernelCodeCap s
       and for all r. GPCapReg s' r = GPCapReg s r
       and if ExceptionLevel s
           then EPCC s' = EPCC s
           else EPCC s' = (PCC s)(Address := PC s + Base (PCC s))
       and for all r. if r ≠ 31 then SpecialCapReg s' r = SpecialCapReg s r
       and for all a. Mem s' a = Mem s a
       and BranchDelay s' = None
       and BranchDelayPCC s' = None
```

3.5.4 The semantics

Before we define the semantics of our abstraction, we define a technical property, which simply states that if the original state is valid, then the resulting state must also be valid: `ValidStateProp s label s' ≡ if StateIsValid s then StateIsValid s'`

We define the semantics of our abstraction as a labelled transition system with signature `State ⇒ (AbstractStep × State) Set`. It allows any execution step that satisfies the properties we defined in this section:

```
(label, s') ∈ AbstractSemantics s ≡
  LoadDataProp s label s'
  and StoreDataProp s label s'
  and ExecuteProp s label s'
  and LoadCapProp s label s'
  and StoreCapProp s label s'
  and StoreLocalCapProp s label s'
  and MemoryInvariant s label s'
  and RestrictCapProp s label s'
  and SealCapProp s label s'
  and UnsealCapProp s label s'
  and SpecialRegisterProp s label s'
  and AddressTranslationProp s label s'
  and CapabilityInvariant s label s'
  and InvokeCapProp s label s'
  and ExceptionProp s label s'
  and ValidStateProp s label s'
```

3.6 Connecting CHERI-MIPS to the abstraction

One of the goals of our abstraction is that it can simulate CHERI-MIPS: an execution step allowed by CHERI-MIPS should also be allowed by our abstraction. Since the abstraction has labelled execution steps of the form `(s, label, s')`, while CHERI-MIPS has execution steps of the form `(s, s')`, we first associate labels with the latter (see Sections 3.6.1–3.6.5). We then state that an execution step allowed by CHERI-MIPS, together with its corresponding label, is allowed by our abstraction (see Section 3.6.6). This is one of the main results of our thesis.

Below we give an overview of mapping an execution step `(s, s')` to a label. Recall that labels have the type `AbstractStep` (see Section 3.4).

- If the execution step has unpredictable behaviour, it maps to `PreserveDomain {}`. The set of abstract actions is empty, because unpredictable behaviour cannot access memory or manipulate capabilities (see Section 2.5.4).
- Otherwise, if the execution step raises a hardware exception, it maps to the abstract step `SwitchDomain RaiseException`.
- Otherwise, if the execution step executes the CCallFast instruction with parameters `cs` and `cb`, it maps to `SwitchDomain (InvokeCapability cs cb)`. The executed instruction can be deduced from `s` by fetching and decoding the next instruction.
- Otherwise, the execution step maps to `PreserveDomain actions`, where the set `actions` depends on the instruction that is executed. There are 56 instructions that access memory or manipulate capabilities, and hence map to non-empty sets of actions. For each of these, we capture its mapping with an auxiliary function that takes the instruction parameters and the starting state as input. For example, `CAndPermActions (cd, cb, rt) s` returns the actions that the CAndPerm (cd, cb, rt) instruction maps to. The state is also a parameter, because in some cases the mapping depends on the TLB or on register values. We explain the most interesting cases in Sections 3.6.1–3.6.4, and the remainder in Appendix A.3.

3.6.1 Restricting capabilities

We map CHERI-MIPS instructions that copy capabilities between registers and/or change their bounds, permissions, or addresses to the `RestrictCapAction` action. For example, the CAndPerm instruction with parameters `cd`, `cb`, and `rt` copies the capability in register `cb` to register `cd`, and restricts the permissions of the copy based on the contents of the general purpose register `rt`. The function `CAndPermActions` captures its mapping:

```
CAndPermActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Writing a capability to register 0 has no effect, because this register always contains the null capability. The mapping above is still correct even if the destination register `cd` is 0, because the null capability can be seen as a restriction of any capability, so in particular it is a restriction of the capability in the source register `cb`.

In some cases, the mapping depends on the state. For example, the CMOVZ instruction with parameters `cd`, `cb`, and `rt` copies the capability in register `cb` to register `cd`, but only if the general purpose register `rt` contains 0. We capture this as follows:

```
CMOVZActions (cd, cb, rt) s ≡
  if GPR s rt = 0
  then {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
  else {}
```

Branch instructions use `RegBranchDelayPCC` (see Section 3.3) instead of `RegPCC` as the destination register, because branches are delayed with one instruction. For example, the CJR instruction maps to the following action:

```
CJRActions cb s ≡
  {RestrictCapAction (RegGeneral cb) RegBranchDelayPCC}
```

The instruction in the branch delay slot completes the branch by copying the capability from the branch delay PCC to the PCC. In Section 3.6.5 we explain how this affects the mapping of that instruction.

Instructions may map to multiple actions with disjoint targets. For example, the CJALR instruction with parameters `cd` and `cb` copies the current PCC to register `cd` and copies the capability in register `cb` to the branch delay PCC. Other examples are the CClearLo and CClearHi instructions that clear the tags of several capabilities at once.

3.6.2 Sealing and unsealing capabilities

Recall that both `SealCapAction` and `UnsealCapAction` have parameters `auth`, `r`, and `r'`, which are respectively the register of the capability that is used as authority, the source register, and the destination register. Here, `auth` has type `CapRegister`, while `r` and `r'` are 5-bit register indices. There are two instructions that map to these actions, namely CSeal and CUnseal. These instructions have no effect if the destination is register 0, because that register always contains the null capability, but otherwise their mapping is straightforward:

```
CSealActions (cd, cs, ct) s ≡
  if cd = 0 then {} else {SealCapAction (RegGeneral ct) cs cd}

CUnsealActions (cd, cs, ct) s ≡
  if cd = 0 then {} else {UnsealCapAction (RegGeneral ct) cs cd}
```

3.6.3 Loading and storing data

Recall that both `LoadDataAction` and `StoreDataAction` have parameters `auth`, `a`, and `l`, which are respectively the register of the capability that is used as authority, the physical address that is accessed, and the length of the access. The parameters of load and store instructions specify the virtual address of the access, which we need to translate before we can instantiate the physical address `a` of the action. For example, consider the CStore instruction with parameters `rs`, `cb`, `rt`, `offset`, and `t`. The function `CStoreVirtualAddress` captures the virtual address that is stored to:

```
CStoreVirtualAddress cb rt offset t s ≡
  Address (GPCapReg s cb) + GPR s rt +
  (SignedCast offset << WordToNat t)
```

Then, `CStorePhysicalAddress` translates this address. This returns an option type, because address translation can fail.

```
CStorePhysicalAddress cb rt offset t s ≡
  TranslateAddr (CStoreVirtualAddress cb rt offset t s, Store) s
```

Finally, the function `CStoreActions` maps the instruction to a set of actions. If the address translation failed, it maps to an empty set. Otherwise, it maps to `StoreDataAction` with the appropriate footprint:

```
1 CStoreActions (rs, cb, rt, offset, t) s ≡
2   case CStorePhysicalAddress cb rt offset t s
3   of None ⇒ {}
4      Some a ⇒ let length = 2^WordToNat t in
5                {StoreDataAction (RegGeneral cb) a length}
```

When mapping legacy accesses, we use the default data capability (DDC) as authority. For example, the mapping of SD refers to the auxiliary function `LegacyStoreActions`:

```
SDActions (b, rt, offset) s ≡
  LegacyStoreActions b offset 8 s
```

and `LegacyStoreActions` specifies `RegSpecial 0` as the authority (recall that the DDC is an alias of special capability register 0):

```
LegacyStoreActions b offset l s ≡
  case LegacyStorePhysicalAddress b offset s
  of None ⇒ {}
     Some a ⇒ {StoreDataAction (RegSpecial 0) a l}
```

The footprints of unaligned accesses are difficult to get right and have caused bugs in the L3 specification (see Chapter 6). Instructions that perform unaligned accesses come in pairs. For example, SDL and SDR store respectively the most- and the least significant part of a doubleword to an unaligned memory address. We make their footprints explicit in the mapping of these instructions:

```
SDLActions (b, rt, offset) s ≡
  let vAddr = LegacyStoreVirtualAddress b offset s in
  let start = vAddr in
  let length = (NOT UnsignedCast vAddr AND Mask 3) + 1 in
  case TranslateAddr (start, Store) s
  of None ⇒ {}
     Some pAddr ⇒ {StoreDataAction (RegSpecial 0) pAddr length}
```

```
SDRActions (b, rt, offset) s ≡
  let vAddr = LegacyStoreVirtualAddress b offset s in
  let start = vAddr AND NOT Mask 3 in
  let length = (UnsignedCast vAddr AND Mask 3) + 1 in
  case TranslateAddr (start, Store) s
  of None ⇒ {}
     Some pAddr ⇒ {StoreDataAction (RegSpecial 0) pAddr length}
```

Our theorems show that these footprints are correct in the sense that SDL and SDR only succeed if the DDC has authority to the footprints.

The examples so far only showed instructions that store data. Instructions that load data are mapped in a similar way.

3.6.4 Loading and storing capabilities

When loading and storing capabilities, our abstraction keeps track of respectively the destination and the source register, which it does not do when accessing data. This gives the following parameters for `LoadCapAction`: the register `auth` of the capability that is used as authority, the physical address `a'` that is loaded from, and the destination register `r`. `StoreCapAction` has similar parameters: the register `auth`, the source register `r`, and the the physical address `a'` that is stored to. In both cases, `a'` is a 35-bit address that identifies a capability-sized and -aligned region. We convert 40-bit addresses to 35-bit addresses by truncating the 5 least significant bits: `GetCapAddress a ≡ Slice 5 a`.

We use the same approach for mapping instructions that access capabilities as for those accessing data. Consider, for example, the mapping of the CSC instruction below:

```
CSCActions (cs, cb, rt, offset) s ≡
  case CSCPhysicalAddress cb rt offset s
  of None ⇒ {}
     Some a ⇒ {StoreCapAction (RegGeneral cb) cs (GetCapAddress a)}
```

Instructions that load capabilities have a special case: if the capability that is used as authority does not have the PermitLoadCapability permission, the load still succeeds, but the tag of the loaded capability is cleared. Because of the missing permission, we cannot map this to `LoadCapAction`. Instead, we map this to a combination of `LoadDataAction` and `RestrictCapAction`. The first action is necessary because the executing code gets access

to the byte representation of the capability. The second action is necessary because the capability in the destination register is changed. We can choose any register as the source register of `RestrictCapAction`, because invalid capabilities are less than or equal to any other capability. For example, consider `CLCActions` below. Line 5 checks whether the PermitLoadCapability permission is present. If so, the instruction is mapped to `LoadCapAction` as normal (Line 6). If not, the instruction maps to the two actions as described above (Lines 7–8).

```
1 CLCActions (cd, cb, rt, offset) s ≡
2   case CLCPhysicalAddress cb rt offset s
3   of None ⇒ {}
4      Some a ⇒
5        if PermitLoadCapability (GPCapReg s cb)
6        then {LoadCapAction (RegGeneral cb) (GetCapAddress a) cd}
7        else {RestrictCapAction (RegGeneral cd) (RegGeneral cd),
8              LoadDataAction (RegGeneral cb) a 32}
```

3.6.5 The entire execution step

To map an arbitrary instruction to a set of abstract actions, we define the function `InstructionActions`. It has parameters `instr` and `s`, where `instr` has type `Instruction`. This type is a tagged union defined in the L3 specification that specifies instructions and their parameters. `InstructionActions` is simply a large case split over `instr`, calling the functions `CAndPermActions`, etc. that we defined above. Its formal definition is included in the appendix (see Definition A.74 on page 178).

We map execution steps in branch delay slots to two additional actions. Recall that at the end of such steps, the capability in `BranchDelayPCC` is copied to the PCC, and `BranchDelayPCC` is cleared. The definition below captures this. It first checks whether the execution step is in a branch delay slot. If not, there are no additional actions, and otherwise, there are two actions that correspond to changing the PCC and clearing the `BranchDelayPCC`:

```
1 BranchDelaySlotActions s ≡
2   case BranchDelayPCC s
3   of None ⇒ {}
4      Some x ⇒
5        {RestrictCapAction RegBranchDelayPCC RegPCC,
6         RestrictCapAction RegBranchDelayPCC RegBranchDelayPCC}
```

The mapping of an execution step depends on the instruction that is executed, and in particular on whether the instruction is CCallFast, which causes a protection domain transition. We define an auxiliary function that provides this information, with the somewhat unwieldy name `FetchCCallFastOrOtherInstruction` to reflect its purpose. It uses `Fetch` and `Decode` that are defined in the L3 specification to fetch and decode

the next instruction. It returns a tagged union type: `NoInstruction` if instruction fetch failed, `CCallFastInstruction cs cb` if the fetched instruction is CCallFast (cs, cb), and `OtherInstruction instr` otherwise, where `instr` is of type `Instruction`. Its formal definition is included in the appendix (see Definition A.76 on page 180).

We formalise the top-level mapping in `MapExecutionStep`, which follows the outline at the beginning of this section. It first checks whether the execution step has unpredictable behaviour (Line 2 below). Recall that `Next` has return type `Unit × State`, so we use the projection `Second` to obtain the resulting state. If the execution step is unpredictable, it returns `PreserveDomain {}` (Line 3), and otherwise it checks whether the execution step raises an exception (Line 4). Recall that `_Next` is the same as `Next`, except it does not clear `ExceptionSignalled` (see Section 2.5.2). If an exception is raised, it returns `SwitchDomain RaiseException` (Line 5), and otherwise it uses the auxiliary function described above to obtain the executed instruction (Line 6). If the instruction is CCallFast (cs, cb), it returns `SwitchDomain (InvokeCapability cs cb)` (Line 8). For other instructions, it combines the sets `InstructionActions` and `BranchDelaySlotActions` and returns `PreserveDomain` of the union (Line 10). If no instruction could be fetched, it returns `PreserveDomain {}` (Line 12).

```
1  MapExecutionStep s ≡
2    if IsUnpredictable (Second (Next s))
3    then PreserveDomain {}
4    else if ExceptionSignalled (Second (_Next s))
5         then SwitchDomain RaiseException
6         else case FetchCCallFastOrOtherInstruction s
7              of CCallFastInstruction cd cd' ⇒
8                   SwitchDomain (InvokeCapability cd cd')
9                 OtherInstruction instr ⇒
10                  PreserveDomain (InstructionActions instr s ∪
11                                  BranchDelaySlotActions s)
12                NoInstruction ⇒ PreserveDomain {}
```

3.6.6 Simulating CHERI-MIPS

Having defined the mapping of execution steps to labels, we can augment the semantics of CHERI-MIPS to a labelled transition system `State ⇒ (AbstractStep × State) Set`. Its definition below uses the mapping to create a label (Line 2) and defines `s'` as the result of `Next` (Line 3). Recall that `Next` indicates whether an execution step is unpredictable, but if that is the case, `UnpredictableNext` should be used to obtain the resulting states (see Section 2.5.4). It therefore checks whether the execution step is unpredictable (Line 4). If so, it combines every state `u` in `UnpredictableNext` with the label (Line 5). If not, it returns a set containing one element, namely the label combined with `s'` (Line 6).

```
SemanticsCheriMips s ≡
  let label = MapExecutionStep s in
  let s' = Second (Next s) in
  if IsUnpredictable s'
  then {(label, u) |u. u ∈ UnpredictableNext s}
  else {(label, s')}
```

The theorem below states that any execution step allowed by CHERI-MIPS is also allowed by our abstraction. In other words, CHERI-MIPS satisfies all the security properties that we defined in Section 3.5. This is one of the main results of our thesis.

Theorem 3.17. Let `(s, label, s')` be a labelled execution step. If this step is allowed by the semantics of CHERI-MIPS, then it is allowed by our abstraction.

```
for all s label s'.
if (label, s') ∈ SemanticsCheriMips s
then (label, s') ∈ AbstractSemantics s
```

We discuss its Isabelle/HOL proof in Chapter 5.

Chapter 4

Reasoning about compartments

We now use our abstraction to reason about memory isolation between compartments. There are many ways to compartmentalise a program: compartments can be disjoint, or share code, data, or both; they can have a hierarchy of trust or be mutually untrusting; and they can be dynamically created, adjusted, and destroyed (see Section 1.3.6). This makes it difficult to give a generally applicable definition of compartments. To avoid this, we consider traces instead (see Section 4.1). Regardless of the compartmentalisation setup, a compartment can only transfer control to another compartment through a domain transition. Therefore, an execution trace that only consists of domain-preserving execution steps must belong to a single compartment. By reasoning about such traces, we do not restrict ourselves to a particular compartmentalisation setup.

To enable reasoning about memory isolation, we determine which capabilities a compartment can access or construct during a domain-preserving trace. We first consider only the starting state, and define *available capabilities* as the capabilities that one can recursively access or construct in this state (see Section 4.2). To express the combined authority of several capabilities at once, we define a *compartment authority* as a set of addresses associated with each capability permission, and we say that the *available authority* is the combined authority of all available capabilities (see Section 4.3). We then prove that the set of available capabilities is monotonic on domain-preserving traces (see Section 4.4). This means that any capability that is accessed or constructed during such a trace was already available in the starting state. We continue by proving capability register and memory invariants on domain-preserving traces. For example, if an address `a` is not writable according to the available authority in a state `s`, then the memory at `a` remains unchanged during any domain-preserving trace that starts at `s`.

As an example of reasoning about memory isolation, we describe a simple, concrete compartmentalisation scenario that isolates a single compartment from the rest of the program (see Section 4.5). This scenario is inspired by the reference monitor example in the CHERI-MIPS documentation [162, §9.4]. We grant the compartment access to a

region of memory and we precisely describe how this region of memory needs to be set up. We then prove that the compartment cannot change any other memory, and that it can only transfer control over the execution to a particular set of exit addresses.

4.1 Traces

In the previous chapter we considered labelled execution steps, where the label has type `AbstractStep`. Here we consider labelled traces. We are not interested in the intermediate states, so we define a trace as a starting state, a list of labels of type `AbstractStep`, and a resulting state. As usual, lists are defined inductively: the base case is the empty list `[]`, and the inductive case is `head :: tail`, with `head` an element and `tail` a list.

Traces are produced by semantics such as `SemanticsCheriMips` and `AbstractSemantics`. We define the predicate `IsTrace` that specifies whether a trace `(s, labels, s')` can be produced by a semantics `sem`. Here the list of labels is backwards: the head of `labels` is the label of the last execution step. We define the predicate by induction on `labels`. In the base case the list of labels is empty, and the predicate requires that the starting state equals the resulting state: `IsTrace sem s [] s' ≡ s = s'`. In the inductive case the list of labels is non-empty, namely `label :: labels`. The predicate requires that there exists an intermediate state `x`, such that `(s, labels, x)` is a trace of `sem` (Line 3), and `(label, s')` is a possible result of executing the semantics in `x` (Line 4).

```
1 IsTrace sem s (label :: labels) s' ≡
2   exists x.
3   IsTrace sem s labels x
4   and (label, s') ∈ sem x
```

We defined `IsTrace` as a predicate to make the induction easier to understand, but we would like to use it as a set. The function `Traces` is conceptually the same as `IsTrace`, but just phrased as a set: `(labels, s') ∈ Traces sem s ≡ IsTrace sem s labels s'`.

We distinguish traces that preserve the protection domain from those that do not. The following function determines whether a single step preserves the domain by inspecting the label (see Section 3.4):

```
PreservesDomain label ≡
  case label
  of PreserveDomain actions ⇒ True
     SwitchDomain action ⇒ False
```

We then lift this definition to lists. A trace with an empty list of labels preserves the domain: `TracePreservesDomain [] ≡ True`. A trace with a non-empty list of labels preserves the domain if all the labels preserve the domain:

```
TracePreservesDomain (label :: labels) ≡
  PreservesDomain label and TracePreservesDomain labels
```

4.2 Available capabilities

The motivation behind our definition of *available capabilities* is to provide a characterisation of the valid capabilities that a (potentially untrusted) compartment can access or construct if it is allowed to execute arbitrary code. This is crucial when reasoning about compartments, as it allows reasoning about which memory locations the compartment can access, whether it can delegate its own capabilities to other compartments, and which addresses in other compartments it can jump to.

The definition of available capabilities only depends on the starting state `s` of a trace. It takes into account all the operations a compartment could potentially perform when it executes arbitrary code, but it does not depend on which operations it actually performs. To show that we indeed took all possible operations into account, we prove in Section 4.4 that the set of available capabilities is monotonic on domain-preserving traces, modulo some assumptions. It follows that any capability that is accessed or constructed during the trace was already available in the starting state. The main assumption here is that the compartment does not have the PermitAccessSystemRegisters permission. With that permission, the compartment can change address translations in such a way that any capability in memory becomes accessible. For that reason alone, compartments that are not completely trusted should not be given that permission.

We now define the set of available capabilities as the capabilities that one can recursively access or construct in a state `s`. The base case is that capabilities in accessible registers of `s` are available in `s`. We then define an inductive case for each capability operation. For example, if an available capability has authority to load a capability from memory, then that capability is also available. Or if an available capability has authority to unseal another available capability `cap`, then the unsealed version of `cap` is also available. We formally define the set of available capabilities below.

Definition 4.1. We define the set `AvailableCaps s` inductively, using the following rules.

The base case. We say that a register `r` of type `CapRegister` (see Section 3.3) is always accessible if it is the PCC, the branch delay PCC, a general purpose capability register, or the special capability register 0 or 1 (the DDC or the TLSC). Other special capability registers are not always accessible because they require the PermitAccessSystemRegisters permission. If `r` is always accessible (Line 2 below) and the capability in `r` is valid (Line 3), then that capability is available (Line 4).

```
1 for all s r.
2 if RegisterIsAlwaysAccessible r
3    and Tag (CapReg s r)
4 then CapReg s r ∈ AvailableCaps s
```

Loading capabilities. If `cap` is an available (Line 2), unsealed (Line 3) capability with the PermitLoadCapability permission (Line 4), `a` is a physical address that is a

translation of a virtual address within the bounds of cap (Line 5), and the capability cap' at address a is valid (Line 6), then cap' is available (Line 7).

```
1 for all s a cap.
2 if cap ∈ AvailableCaps s
3    and not IsSealed cap
4    and PermitLoadCapability cap
5    and a ∈ TranslateCapAddresses (RegionOfCap cap) Load s
6    and Tag (MemCap s a)
7 then MemCap s a ∈ AvailableCaps s
```

Restricting capabilities. If cap is an available capability (Line 2), and cap' is a valid capability less than or equal to cap (Lines 3–4), then cap' is available (Line 5).

```
1 for all s cap cap'.
2 if cap ∈ AvailableCaps s
3    and cap' ≤ cap
4    and Tag cap'
5 then cap' ∈ AvailableCaps s
```

Sealing capabilities. If cap is an available (Line 2), unsealed (Line 3) capability, and sealer is an available (Line 4), unsealed capability (Line 5) with the PermitSeal permission (Line 6), and the object type t lies within its bounds (Line 7), then the capability that is the result of sealing cap with object type t is available (Line 8).

```
1 for all s t cap sealer.
2 if cap ∈ AvailableCaps s
3    and not IsSealed cap
4    and sealer ∈ AvailableCaps s
5    and not IsSealed sealer
6    and PermitSeal sealer
7    and UnsignedCast t ∈ RegionOfCap sealer
8 then cap(IsSealed := True, ObjectType := t) ∈ AvailableCaps s
```

Unsealing capabilities. If cap is an available (Line 2), sealed (Line 3) capability, and unsealer is an available (Line 4), unsealed (Line 5) capability with the PermitUnseal permission (Line 6), and the object type of cap lies within its bounds (Line 7), then the capability that is the result of unsealing cap is available (Line 8).

```
1 for all s cap unsealer.
2 if cap ∈ AvailableCaps s
3    and IsSealed cap
4    and unsealer ∈ AvailableCaps s
5    and not IsSealed unsealer
6    and PermitUnseal unsealer
7    and UnsignedCast (ObjectType cap) ∈ RegionOfCap unsealer
8 then cap(IsSealed := False, ObjectType := 0) ∈ AvailableCaps s
```

4.3 Compartment authorities

To express the combined authority of all available capabilities, we first define *compartment authorities*. A compartment authority contains a separate memory region for each capability permission. For example, it can simultaneously express the authority to write data to the entire address space and the authority to write capabilities to only a subset. A capability cannot express this, as all its permissions apply to a single region of memory. The fields of a compartment authority `c` are given below.

`ExecutableAddresses c`
: The set of virtual addresses that can be executed.

`LoadableAddresses c`
: The set of virtual addresses where data can be loaded from.

`CapLoadableAddresses c`
: The set of virtual addresses where capabilities can be loaded from.

`StorableAddresses c`
: The set of virtual addresses where data can be stored to.

`CapStorableAddresses c`
: The set of virtual addresses where capabilities can be stored to.

`LocalCapStorableAddresses c`
: The set of virtual addresses where local capabilities can be stored to.

`SealableTypes c`
: The set of object types that can be sealed.

`UnsealableTypes c`
: The set of object types that can be unsealed.

`SystemRegisterAccess c`
: A boolean indicating whether system registers can be accessed.

We define the function `GetAuthority` to translate capabilities to compartment authorities. If a capability `cap` is valid and has the PermitExecute permission, then the `ExecutableAddresses` field of the authority equals the memory region of `cap`, and otherwise it equals the empty set. The other fields of the authority are defined similarly (see Figure 4.1 on the next page). Note that we ignore whether `cap` is sealed or not.

As mentioned above, compartment authorities can express the authority of multiple capabilities at once. We define the union `c ∪ c'` of two authorities by taking the disjunction or union of their fields. For example:

```
ExecutableAddresses (c ∪ c') = ExecutableAddresses c ∪ ExecutableAddresses c'
```

We also define the big union `⋃cs`, which takes a set `cs` of authorities, and returns their union. For example, we have `a ∈ ExecutableAddresses ⋃cs` if there exists a `c ∈ cs` with `a ∈ ExecutableAddresses c`.

```
GetAuthority cap ≡
  SystemRegisterAccess = Tag cap and PermitAccessSystemRegisters cap
  ExecutableAddresses = if Tag cap and PermitExecute cap
                        then RegionOfCap cap
                        else {}
  LoadableAddresses = if Tag cap and PermitLoad cap
                      then RegionOfCap cap
                      else {}
  CapLoadableAddresses = if Tag cap and PermitLoadCapability cap
                         then RegionOfCap cap
                         else {}
  StorableAddresses = if Tag cap and PermitStore cap
                      then RegionOfCap cap
                      else {}
  CapStorableAddresses = if Tag cap and PermitStoreCapability cap
                         then RegionOfCap cap
                         else {}
  LocalCapStorableAddresses = if Tag cap and PermitStoreLocalCapability cap
                              then RegionOfCap cap
                              else {}
  SealableTypes = if Tag cap and PermitSeal cap
                  then {t. UnsignedCast t ∈ RegionOfCap cap}
                  else {}
  UnsealableTypes = if Tag cap and PermitUnseal cap
                    then {t. UnsignedCast t ∈ RegionOfCap cap}
                    else {}
```

Figure 4.1: The compartment authority that corresponds to a capability `cap`. Each field of the authority is defined with a separate equation. Note that the fields `SealableTypes` and `UnsealableTypes` are sets of 24-bit object types, so we cast the virtual addresses in the memory region of `cap` before using them as object types.

This allows us to define the *available authority*, which captures the authority of a compartment during a domain-preserving trace when it is allowed to execute arbitrary code. We define it by combining the authority of all unsealed, available capabilities into a single compartment authority:

```
AvailableAuthority s ≡
  ⋃{GetAuthority cap |cap. cap ∈ AvailableCaps s and not IsSealed cap}
```

For example, `a ∈ ExecutableAddresses (AvailableAuthority s)` means that there exists a valid, unsealed capability `cap` that has the PermitExecute permission, and that is available in state `s`.

4.4 Security properties about traces

The security properties that we prove here hold for any semantics that can be simulated by our abstraction, not just for CHERI-MIPS. In other words, we rely on the security properties defined in Section 3.5, but not on any other behaviour of CHERI-MIPS. To be able to state that a labelled transition system `sem` can be simulated by our abstraction, we define the following function. Theorem 3.17 shows that this holds for `sem = SemanticsCheriMips`.

```
CanBeSimulated sem ≡
  for all s label s'.
  if (label, s') ∈ sem s
  then (label, s') ∈ AbstractSemantics s
```

First, we prove that each valid capability that is obtained during a domain-preserving execution step was already available in the starting state `s`. This only holds if the PCC in `s` does not have the PermitAccessSystemRegisters permission.

Lemma 4.2 (New capabilities are available)**.** Let `sem` be a labelled transition system that can be simulated by our abstraction (Line 2). Let `s` be a valid state (Line 3), and assume that the PCC does not have the PermitAccessSystemRegisters permission (Line 4). Consider an execution step from `s` to `s'` that preserves the domain (Line 5), and consider a location `loc` that contains a valid capability in the resulting state (Line 6) and a different capability in the original state (Line 7). Then, the new capability was already available in the starting state `s` (Line 8).

```
1 for all sem s s' actions loc.
2 if CanBeSimulated sem
3    and StateIsValid s
4    and not PermitAccessSystemRegisters (PCC s)
5    and (PreserveDomain actions, s') ∈ sem s
6    and Tag (Cap s' loc)
7    and Cap s' loc ≠ Cap s loc
8 then Cap s' loc ∈ AvailableCaps s
```

Then we prove that the set of available capabilities is monotonic during a domain-preserving execution step `(s, label, s')`. In other words, if a capability is available in `s'`, then it was already available in `s`.

Lemma 4.3. Let `sem` be a labelled transition system that can be simulated by our abstraction (Line 2). Let `s` be a valid state (Line 3), and assume that the PCC does not have the PermitAccessSystemRegisters permission (Line 4). Consider an execution step from `s` to `s'` that preserves the domain (Line 5). Then the capabilities that are available in `s'` are also available in `s` (Line 6).

```
1 for all sem s s' actions.
2 if CanBeSimulated sem
3    and StateIsValid s
4    and not PermitAccessSystemRegisters (PCC s)
5    and (PreserveDomain actions, s') ∈ sem s
6 then AvailableCaps s' ⊆ AvailableCaps s
```

We extend the lemma above to domain-preserving traces. The assumption about the PCC in `s` no longer suffices because other capabilities might be used as the PCC during the trace. Instead, we assume that the available authority in `s` does not have system register access. Recall from Section 4.3 that this means that there is no valid, unsealed capability that is available in `s` and that has the PermitAccessSystemRegisters permission.

Theorem 4.4 (Monotonicity of available capabilities)**.** Let `sem` be a labelled transition system that can be simulated by our abstraction (Line 2). Let `s` be a valid state (Line 3), and assume that the available authority does not have system register access (Line 4). Consider an execution trace from `s` to `s'` (Line 5) that preserves the domain (Line 6). Then the capabilities that are available in `s'` are also available in `s` (Line 7).

```
1 for all sem s s' labels.
2 if CanBeSimulated sem
3    and StateIsValid s
4    and not SystemRegisterAccess (AvailableAuthority s)
5    and (labels, s') ∈ Traces sem s
6    and TracePreservesDomain labels
7 then AvailableCaps s' ⊆ AvailableCaps s
```

We then prove three invariants that state, respectively, when special capability registers, data in memory, and capabilities in memory remain unchanged during a trace. Here we consider traces that preserve the domain except for their last execution step, which switches domains. In other words, we consider traces up to the point where another compartment has control. The first invariant states that special capability registers, except for the DDC, TLSC, and EPCC, remain unchanged if the available authority does not have system register access. The EPCC is excluded because the last execution step might raise an exception, which may change the EPCC.

Theorem 4.5 (Special capability register invariant)**.** Let sem be a labelled transition system that can be simulated by our abstraction (Line 2). Let s be a valid state (Line 3), and assume that the available authority does not have system register access (Line 4). Consider a special capability register index r that does not point to the DDC, TLSC, or EPCC (Line 5). Then consider an execution trace from s to s' (Line 6) that preserves the domain up to the last execution step (Line 7), but that switches domains at the last step (Line 8). Then the special capability register r contains the same capability in s and s' (Line 9).

```
1 for all s s' labels label r.
2 if CanBeSimulated sem
3    and StateIsValid s
4    and not SystemRegisterAccess (AvailableAuthority s)
5    and r ≠ 0 and r ≠ 1 and r ≠ 31
6    and (label :: labels, s') ∈ Traces sem s
7    and TracePreservesDomain labels
8    and not PreservesDomain label
9 then SpecialCapReg s' r = SpecialCapReg s r
```

The second invariant states that the memory at an address a remains unchanged if the available authority does not have system register access, and a is not a translation of an address that is storable according to the available authority.

Theorem 4.6 (Memory invariant)**.** Let sem be a labelled transition system that can be simulated by our abstraction (Line 2). Let s be a valid state (Line 3), and assume that the available authority does not have system register access (Line 4). Let virtual be the set of virtual addresses that are storable according to the available authority (Line 5) and assume that the address a is not a translation of any of these addresses (Line 6). Then consider an execution trace from s to s' (Line 7) that preserves the domain up to the last execution step (Line 8), but that switches domains at the last step (Line 9). Then the memory at a contains the same value in s and s' (Line 10).

```
1  for all s s' labels label a.
2  if CanBeSimulated sem
3     and StateIsValid s
4     and not SystemRegisterAccess (AvailableAuthority s)
5     and let virtual = StorableAddresses (AvailableAuthority s) in
6         not a ∈ TranslateAddresses virtual Store s
7     and (label :: labels, s') ∈ Traces sem s
8     and TracePreservesDomain labels
9     and not PreservesDomain label
10 then MemByte s' a = MemByte s a
```

The third invariant states when capabilities in memory remain unchanged. This is not a consequence of the invariant above, as that invariant does not consider memory

tags. Recall that `GetCapAddress` converts a 40-bit physical address to a 35-bit address that denotes the corresponding capability-sized and -aligned region. The invariant below considers a 35-bit address `a` and assumes that there is no virtual address that is storable according to the available authority, and that translates to a physical address `a'` with `a = GetCapAddress a'`.

Theorem 4.7 (Memory tag invariant)**.** Let `sem` be a labelled transition system that can be simulated by our abstraction (Line 2). Let `s` be a valid state (Line 3), and assume that the available authority does not have system register access (Line 4). Let `virtual` be the set of virtual addresses that are storable according to the available authority (Line 5), let `physical` be the translation of these addresses, and assume that the address `a` does not equal `GetCapAddress a'` for any `a'` in this set of translations (Line 7). Then consider an execution trace from `s` to `s'` (Line 8) that preserves the domain up to the last execution step (Line 9), but that switches domains at the last step (Line 10). Then the capability-sized and -aligned region of memory at `a` contains the same capability in `s` and `s'` (Line 11).

```
1  for all s s' labels label a.
2  if CanBeSimulated sem
3     and StateIsValid s
4     and not SystemRegisterAccess (AvailableAuthority s)
5     and let virtual = StorableAddresses (AvailableAuthority s) in
6         let physical = TranslateAddresses virtual Store s in
7         not a ∈ {GetCapAddress a' |a'. a' ∈ physical}
8     and (label :: labels, s') ∈ Traces sem s
9     and TracePreservesDomain labels
10    and not PreservesDomain label
11 then MemCap s' a = MemCap s a
```

4.5 A simple compartmentalisation scenario

Finally, we consider a simple compartmentalisation scenario, where a compartment is isolated from the rest of the program. Isolation here means that the compartment can only access its own region of memory, cannot access special capability registers, and when it yields the execution it can jump only to a restricted set of addresses. To make this more precise, let `addresses` be the set of addresses that we allow the compartment to access (as code or data), and let `exit` be the set of addresses we allow it to jump to. We consider a trace from state `s` to `s'` that preserves the domain up to the last execution step, and that switches domains at the last step. The predicate below formalises the isolation guarantees.

Definition 4.8. The predicate `IsolationGuarantees` holds for `addresses`, `exit`, `s`, and `s'` if all of the following conditions hold: the address of the instruction that is executed in `s'`

is contained in exit (Line 2); for all special capability register indices r that do not point to the DDC, TLSC, or EPCC (Line 4), the register r contains the same capability in s and s' (Line 5); and for all physical addresses a that are not the translation of an address in addresses (Line 7), the memory at a contains the same value in s and s' (Line 8), and the tag of the corresponding capability-sized and -aligned region is the same in s and s' (Line 9).

```
1 IsolationGuarantees addresses exit s s' ≡
2   Base (PCC s') + PC s' ∈ exit
3   and for all r.
4       if r ≠ 0 and r ≠ 1 and r ≠ 31
5       then SpecialCapReg s' r = SpecialCapReg s r
6   and for all a.
7       if not a ∈ TranslateAddresses addresses Store s
8       then MemByte s' a = MemByte s a
9            and MemTag s' (GetCapAddress a) = MemTag s (GetCapAddress a)
```

CHERI only guarantees this if the compartment is set up correctly. There is one main and two minor requirements. The first minor requirement is that the addresses of exception handlers should be allowed exit points, because the compartment might raise an exception. In other words, ExceptionPCs ⊆ exit. The other minor requirement is that the set of addresses that we allow the compartment to access should be capability-aligned. In other words, if we allow the compartment to access an address a, and an address b lies in the same capability-sized and -aligned region of memory, then we should allow the compartment to access b:

```
CapabilityAligned addresses ≡
  for all a b.
  if a ∈ addresses and a AND NOT Mask 5 = b AND NOT Mask 5
  then b ∈ addresses
```

Note that addresses does not need to be contiguous.

The main requirement is that the authority of the compartment does not extend beyond addresses. One could define this in terms of AvailableAuthority s, but this depends on AvailableCaps, whose inductive definition is difficult to work with. Instead, we define the requirement in terms of the capabilities that are granted to the compartment. These capabilities are present in the registers and memory of s, namely the capabilities in always-accessible registers (Line 2 below), and the capabilities at an address a that is a translation of an address in addresses (Line 3):

```
1 GrantedCaps addresses s ≡
2   {CapReg s r |r. RegisterIsAlwaysAccessible r} ∪
3   {MemCap s (GetCapAddress a) |a. a ∈ TranslateAddresses addresses Load s}
```

We consider two subsets of the granted capabilities. The first is the set of capabilities that can be invoked or that can be transformed into a capability that can be invoked. We

define them as the valid, granted capabilities that have the PermitCCall permission:

```
cap ∈ InvokableCaps addresses s ≡
  cap ∈ GrantedCaps addresses s and Tag cap and PermitCCall cap
```

The second is the set of capabilities that the compartment can use to authorise memory accesses or capability manipulations. The compartment can use a valid, granted capability if it is either unsealed, or sealed with an object type that the compartment can unseal. With `types` the set of object types that are granted to the compartment, we capture this as follows:

```
cap ∈ UsableCaps addresses types s ≡
  cap ∈ GrantedCaps addresses s and Tag cap
  and not IsSealed cap or ObjectType cap ∈ types
```

We then formulate requirements on the sets defined above. The first requirement ensures that the compartment cannot access special capability registers, change address translations, or perform other privileged operations. It requires that none of the usable capabilities may have the PermitAccessSystemRegisters permission:

```
NoSystemRegisterAccess addresses types s ≡
  for all cap.
  if cap ∈ UsableCaps addresses types s
  then not PermitAccessSystemRegisters cap
```

The second requirement ensures that the compartment cannot access any memory outside the set `addresses` that we allow the compartment to access. It considers usable capabilities that can authorise a memory access (of any kind), and requires that their memory region is contained in `addresses`:

```
ContainedCapBounds addresses types s ≡
  for all cap.
  if cap ∈ UsableCaps addresses types s
     and PermitExecute cap
         or PermitLoad cap
         or PermitLoadCapability cap
         or PermitStore cap
         or PermitStoreCapability cap
         or PermitStoreLocalCapability cap
  then RegionOfCap cap ⊆ addresses
```

The third requirement ensures that the compartment cannot seal or unseal object types other than those in `types`. It considers usable capabilities that have the PermitSeal or PermitUnseal permission, and requires that any 24-bit object type whose 64-bit cast is contained in the capability's memory region, is contained in `types`:

```
ContainedObjectTypes addresses types s ≡
  for all cap t.
  if cap ∈ UsableCaps addresses types s
     and PermitSeal cap or PermitUnseal cap
     and UnsignedCast t ∈ RegionOfCap cap
  then t ∈ types
```

The final requirement ensures that the compartment cannot use any of the invokable capabilities, and that the invokable capabilities only point to allowed exit points. More precisely, it requires that each invokable capability is sealed with an object type not in `types`, and that its address is contained in `exit`:

```
InvokableCapsSetup addresses types exit s ≡
  for all cap.
  if cap ∈ InvokableCaps addresses s
  then IsSealed cap and ObjectType cap ∉ types and Address cap ∈ exit
```

We collect these requirements by taking their conjunction:

```
CapabilitySetup addresses types exit s ≡
  NoSystemRegisterAccess addresses types s
  and ContainedCapBounds addresses types s
  and ContainedObjectTypes addresses types s
  and InvokableCapsSetup addresses types exit s
```

The theorem below captures the assumptions and guarantees of the compartmentalisation scenario.

Theorem 4.9. Let `sem` be a labelled transition system that can be simulated by our abstraction (Line 2). Let `s` be a valid state (Line 3), assume that the set `addresses` is capability-aligned (Line 4), assume that all exception handlers are considered valid exit points (Line 5), and assume that the capabilities of the compartment are set up correctly (Line 6). Furthermore, consider an execution trace from `s` to `s'` (Line 7) that preserves the domain up to the last execution step (Line 8), but that switches domains at the last step (Line 9). Then the isolation guarantees as defined in Definition 4.8 on page 121 hold (Line 10).

```
for all sem addresses types exit s s' labels label.
if CanBeSimulated sem
   and StateIsValid s
   and CapabilityAligned addresses
   and ExceptionPCs ⊆ exit
   and CapabilitySetup addresses types exit s
   and (label :: labels, s') ∈ Traces sem s
   and TracePreservesDomain labels
   and not PreservesDomain label
then IsolationGuarantees addresses exit s s'
```

Chapter 5

The Isabelle proof development

In Section 1.5.2 we described three challenges in proving security properties for production-scale architectures: these architectures are large, their low-level details are easy to miss, and they typically keep evolving. We solve these challenges by mechanising our proofs in Isabelle/HOL, developing automated proof tactics, and generating repetitive parts of the proofs with Python scripts. As described before, our proof tactics and Python scripts do not need to be trusted, because their output is verified by Isabelle's LCF-style inference kernel.

Pen-and-paper proofs typically focus on the conceptually interesting parts of a proof, providing insight in the reason why the statement is true. The fact that mechanised proofs cannot skip over uninteresting details is crucial to give confidence that our security properties hold, but it also makes the proofs less accessible and less maintainable. To alleviate this, we develop automated proof tactics that can prove these uninteresting parts (see Section 5.1).

We then give an overview of our proof development, including our Python scripts (see Section 5.2). The entire proof is 32k lines of Isabelle/HOL, of which 14k lines are generated. Verifying all these proofs takes 9 minutes on a 24GB Intel i7-9700K. The Isabelle source files and the Python scripts are available online [108].

5.1 Automated proof tactics

Isabelle has powerful automated proof tactics that we can use to prove conceptually uninteresting parts of our proofs. Unfortunately, because of the scale of our proof goals, these tactics can get lost in a combinatorial explosion. To avoid this, we develop custom tactics that guide Isabelle's tactics through the proof goal. We use Hoare logic [51, 65] to achieve this, combined with lemmas about the footprints of the auxiliary functions in the CHERI-MIPS specification.

To explain how we arrived at our proof tactics, we first describe how a straightforward application of Isabelle's tactics leads to a combinatorial explosion. We then iteratively refine our approach to avoid this problem, leading to the Hoare logic proof tactic (see Section 5.1.5) that is the backbone of our proofs.

5.1.1 Straightforward expansion

In our first approach, to prove some desired statement, we would expand the definitions in the CHERI-MIPS specification that the statement depends on, including the definition of the state monad, until we are left with an expression that directly reads and updates the CHERI-MIPS machine state. We would then use *auto*, Isabelle's default automated proof tactic, to prove the statement.

We illustrate the approach on a simple, but non-trivial lemma about the DADD instruction, which performs a 64-bit signed addition. This lemma is made up for illustration purposes only, and does not play a role in our security proofs. It states that adding a GPR to itself does not change its sign, which is true in MIPS because DADD raises an exception if the addition would overflow, and exceptions do not change any GPRs.

Lemma 5.1. Let `s'` be the resulting state of applying `ExecuteDADD` to the register indices `(i, i, i)` in state `s`. Then the sign of GPR `i` is the same in `s'` as in `s`. Formally:

```
for all s i.
let s' = Second (ExecuteDADD (i, i, i) s) in
Bit (GPR s' i) 63 = Bit (GPR s i) 63
```

Recall that `ExecuteDADD` is a monadic function with return type `Unit` × `State`, so to obtain only the resulting state we use the projection function `Second`. Also note that `ExecuteDADD` does not describe an entire execution step: it is an auxiliary function that describes the part of the execution step that is specific to the DADD instruction (see Figure 2.16 on page 77). It excludes, for example, instruction fetch, decoding, and handling delayed branches. The example lemma excludes these because straightforward expansion is not a feasible approach there.

To prove the lemma, we expand the definitions of `ExecuteDADD` and `GPR`, and the functions they (transitively) depend on, such as `SignalException` (see Appendix A.2 on pages 164–168). We also expand the part of the state monad it depends on, namely `Return`, `ReadState`, `UpdateState`, `Bind`, `ExtendState`, and `TrimState`. This presents the following problem:

Combinatorial explosion. The definitions of `Bind`, `ExtendState`, and `TrimState` use a let statement `let (v, s) = f in g`, where `f` and `g` are expressions. Expanding the let statement, which Isabelle's proof tactic *auto* may attempt, duplicates `f` for each occurrence of `v` and `s` in `g`. With nested let statements this may lead to a combinatorial explosion:

consider `let (v, s) = f in let (v', s') = g in h`, where `g` contains `k` occurrences of `v` and `s`, `h` contains `m` occurrences of `v` and `s`, and `h` contains `n` occurrences of `v'` and `s'`. Then expanding the let statements duplicates `f`, `(k * n + m)` times.

Expanding Lemma 5.1 results in a statement with 189 nested let statements. Isabelle's proof tactic *auto* is able to prove the lemma, albeit slowly: it takes around 10 seconds (on a 24GB Intel i7-9700K). However, this approach quickly becomes infeasible. Consider, for example, the following lemma about the CSetBounds instruction, which creates a copy of a capability and restricts the bounds of the copy. The lemma states that either the original capability has a tag, or an exception is raised:

Lemma 5.2. Let `s'` be the resulting state of applying `ExecuteCSetBounds` to the indices `(cd, cb, rt)` in state `s`. Then either the source capability `GPCapReg cb s` has a tag, or `ExceptionSignalled` is set in `s'`. Formally:

```
for all s cd cb rt.
let s' = Second (ExecuteCSetBounds (cd, cb, rt) s) in
Tag (GPCapReg s cb) or ExceptionSignalled s'
```

Expanding this lemma results in a statement with 973 nested let statements, which *auto* can no longer prove within reasonable time (we aborted *auto* after several minutes). The reason for the number of let statements is that CSetBounds has five conditions under which it can raise an exception, so expanding the lemma duplicates the definition of `SignalException` five times, worsening the combinatorial explosion.

5.1.2 Lemmas about state changes

Our second approach exploits the following: our security properties put requirements on specific parts of the machine state, such as capability registers, exception flags, and the memory, but not on the machine state as a whole. In our second approach, we would state how definitions in the CHERI-MIPS specification change these parts of the state. For example, the lemmas below state that `SignalException` does not change `MemCap`, and sets `ExceptionSignalled` to true.

Lemma 5.3. Let `s'` be the resulting state of applying `SignalException` to the exception type `t` in state `s`, and let `a` be a 35-bit address. Then the capability at `a` in `s'` is the same as in `s`. Formally:

```
for all s t a.
let s' = Second (SignalException t s) in
MemCap s' a = MemCap s a
```

Lemma 5.4. Let `s'` be the resulting state of applying `SignalException` to the exception type `t` in state `s`. Then `ExceptionSignalled` is true in `s'`. Formally:

```
for all s t.
let s' = Second (SignalException t s) in
ExceptionSignalled s' = True
```

In some cases, these lemmas remove the need to expand a definition, alleviating the combinatorial explosion. For example, using Lemma 5.4 and a similar lemma about `SignalCapException`, we can prove Lemma 5.2, which our previous approach could not prove.

However, it is impractical to state a lemma of the form `X (Second (Y s)) = ...` for each combination of a state part `X` and a function `Y`, even if we restrict `Y` to low-level auxiliary functions. First, this would require thousands of lemmas: our security properties directly refer to 16 state parts, and indirectly depend on a factor more, while there are around 120 relevant low-level auxiliary functions. Second, for some combinations even stating the value `X (Second (Y s))` would be unproductive, as it would duplicate a significant part of the definition of `Y`. For example, consider the function `StoreMemoryCap` that stores data (contrary to what its name suggests), and the state part `MemCap` that returns the capability at a 35-bit address `a`. We informally describe the value `MemCap (Second (StoreMemoryCap v s)) a` below to give an impression of its complexity, but the details do not need to be understood:

- If the virtual address `vAddr` that is specified in `v` cannot be translated, then the value equals `MemCap s a`.
- Else, if `v` specified that the store should be aligned, but the physical address `pAddr` that `vAddr` translates to is not aligned, then the value equals `MemCap s a`.
- Else, if the upper bits of `pAddr` equal the base address of the JTAG UART, then the value equals `MemCap s a`.
- Else, if `pAddr` is in use by the PIC of one of the processor cores, then the value equals `MemCap s a`.
- Else, if `v` specified that the store is conditional and the `LoadLinkFlag` is not set, is `False`, or is set with a different address, then the value equals `MemCap s a`.
- Else, if the footprint `pAddr`, ..., `pAddr + length - 1` does not overlap with the capability at `a`, then the value equals `MemCap s a`.
- Else, the value equals an invalid capability whose byte representation is given by the old memory contents overwritten with the new value specified in `v`.

Stating this formally would duplicate a significant part of the definition of `StoreMemoryCap`.

This means that our second approach on its own is not sufficient to prevent combinatorial explosions.

5.1.3 Hoare logic

In our third approach we use Hoare logic [51, 65] to expand the definition of the state monad step by step, instead of expanding it all at once. This allows us to remove duplications as they arise, preventing a combinatorial explosion in many cases. We describe the details of this approach below.

To prove some desired statement using Hoare logic, we would first reformulate the statement as a Hoare triple `pre`, `m`, `post`, where `pre` is a condition on states, `m` is a monadic computation, and `post` is a condition on both the resulting value and the resulting state [103, 80]. The triple is true if for each state `s` that satisfies `pre`, running `m` in `s` results in a value and state that satisfies `post`. We prefix the formal definition `_HoareTriple` below with an underscore because we define another variant of Hoare triples in a later approach.

Definition 5.5. Let `pre` be a condition of type `State ⇒ Bool`, `m` a monadic function of type `State ⇒ 'a × State`, and `post` a condition of type `'a ⇒ State ⇒ Bool`. Then the Hoare triple `pre`, `m`, `post` holds if for all `s` with `pre s` we have `post value s'`, where `value` and `s'` are respectively the resulting value and state of `m s`. Formally:

```
_HoareTriple pre m post ≡
  for all s.
  let (value, s') = m s in
  if pre s then post value s'
```

As an example, we reformulate Lemma 5.1 to a Hoare triple. Because the conclusion of the lemma refers to both the starting state and the resulting state but postconditions only have the resulting state as a parameter, we cannot directly use the conclusion as the postcondition. To circumvent this, we introduce a new variable `oldBit`, assume in the precondition that the sign of GPR `i` in the starting state equals `oldBit`, and require in the postcondition that the sign of GPR `i` still equals `oldBit` in the resulting state. The reformulated lemma below is equivalent to Lemma 5.1.

Lemma 5.6. Let `i` be a register index and `oldBit` a sign. Let `inv` be the function that takes a state `s` as input and returns whether the sign of GPR `i` in `s` equals `oldBit`. Then `inv` is an invariant of `ExecuteDADD (i, i, i)`. In other words, `inv`, `ExecuteDADD (i, i, i)`, `λv. inv` is a Hoare triple. Formally:

```
1 for all i oldBit.
2 _HoareTriple
3   (λs. Bit (GPR s i) 63 = oldBit)
4   (ExecuteDADD (i, i, i))
5   (λv s'. Bit (GPR s' i) 63 = oldBit)
```

Then, to prove Hoare triples, we develop a custom proof tactic in Eisbach [87], an extension of Isabelle's proof language. Proof tactics operate on proof goals, discharging

them or replacing them with new goals by applying lemmas or other proof tactics. However, we find it more intuitive to explain our proof tactic as an algorithm that produces an output based on inputs. For example, our Hoare proof tactic expects a proof goal `_HoareTriple (pre, m, post)`, where `pre` is a free variable, and `m` and `post` expressions, and it then applies lemmas, other tactics, and calls itself recursively, until all proof goals are discharged. In the process, the free variable `pre` has been instantiated to a more specific expression `pre'`. Informally, we would describe this as if the tactic "constructs" a precondition `pre'`. In this thesis, we describe our proof tactics in this informal style, and we refer to the Isabelle source [108] for the Eisbach implementations of the tactics we use in our proofs.

Our Hoare proof tactic takes a monadic expression `m` and a postcondition `post` as input, and constructs a precondition `pre` such that `pre`, `m`, `post` is a Hoare triple. It is defined recursively on the structure of `m`. We describe the cases that are relevant to our discussion below:

1. If `m` has the form `Return x`, `ReadState f`, or `UpdateState g`, our tactic simply expands the monadic definition, which leads to respectively the preconditions `post x`, `λs. post (f s) s`, or `λs. post () (g s)`. This is similar to our first approach (see Section 5.1.1) and likewise suffers from a similar problem: with `v` and `s` the parameters of `post`, the expressions `x` and `f` are duplicated for each occurrence of `v` in `post`, and `g` is duplicated for each occurrence of `s` in `post`.
2. Similarly, if `m` refers to an auxiliary function, our tactic constructs the precondition `λs. post (First (m s)) (Second (m s))`, duplicating `m` for each occurrence of either parameter in `post`.
3. The recursive case where `m` has the form `Bind m' n'` is crucial to prevent a combinatorial explosion. Our tactic traverses `m` backwards: first, it recursively constructs a precondition of `n'` and `post`, which we call `intermediate`. It then invokes *simp*, which is a term-rewriting tactic in Isabelle, to obtain a simpler, but equivalent function `intermediate'`. We supply the *simp* tactic with some of the lemmas of the previous approach, namely invariants `X (Second (Y s)) = X s`, such as Lemma 5.3, and lemmas of the form `X (Second (Y s)) = C` where `C` is some constant, such as Lemma 5.4. For the moment, we ignore that this would require an impractical number of lemmas. Finally, the tactic recursively constructs a precondition of `m'` and `intermediate'`, which it uses as the precondition of `Bind m' n'` and `post`.

In most cases, our proof tactic avoids a combinatorial explosion: by expanding the definition of the state monad step by step, we can simplify intermediate results, which prevents duplications at later steps.

However, if the *simp* tactic is unable to simplify an expression `X (Second (Y s))` with `X` a state part and `Y` an auxiliary function, then lemmas of the form `X (Second (Z s)) = ...`

for other functions `Z` no longer apply, obstructing our mechanism to prevent combinatorial explosions. For example, consider the postcondition `MemCap s a = cap` and the following (contrived) monadic expression:

```
1 do {
2   x ← WritePCC p;
3   x ← WriteGPCapReg v;
4   WriteData w
5 }
```

When our proof tactic has processed `WriteData w`, the constructed precondition equals `MemCap (Second (WriteData w s)) a = cap`, for which we do not have a simplification lemma. Our proof tactic then processes `WriteGPCapReg v`. We do have the simplification lemma `MemCap (Second (WriteGPCapReg v s)) a = MemCap s a`, but our proof tactic cannot apply it because `WriteData` is in the way. The constructed precondition thus becomes the unwieldy `MemCap (Second (WriteData w (Second (WriteGPCapReg v s)))) a = cap`. Similarly, we cannot apply the lemma `MemCap (Second (WritePCC p s)) a = MemCap s a` because now both `WriteData` and `WriteGPCapReg` are in the way, complicating the precondition further.

5.1.4 Commutativity

We refine the previous approach by exploiting the fact that many auxiliary functions *commute*: changing their order does not change the result. In many cases, commutativity allows us to circumvent the problem we described above, where a function blocks the application of simplification lemmas. Repeating the supporting example, `WriteData` blocks the application of the lemma `MemCap (Second (WriteGPCapReg v s)) a = MemCap s a` in Line 1 below. Because `WriteGPCapReg` commutes with `WriteData`, we can rewrite Line 1 to Line 2. As a consequence, `WriteData` no longer blocks the simplification lemma, so we can rewrite Line 2 to Line 3.

```
1 MemCap (Second (WriteData w (Second (WriteGPCapReg v s)))) a =
2 MemCap (Second (WriteGPCapReg v (Second (WriteData w s)))) a =
3 MemCap (Second (WriteData w s)) a
```

We describe the details of this approach below, starting with the definition of commutativity.

Definition 5.7. The monadic functions `m` and `n` commute if first applying `m` and then applying `n` gives the same resulting state and resulting values as when applying them in the reverse order. Formally:

```
Commute m n ≡
  do {
    v ← m;
    w ← n;
    Return (v, w)
  } = do {
    w ← n;
    v ← m;
    Return (v, w)
  }
```

For each low-level auxiliary function `Y` we prove a lemma that states when `Y` commutes with an arbitrary monadic function `n`. First, we determine the *footprint* of `Y`, which consists of the auxiliary functions $\texttt{Z}_1, \ldots, \texttt{Z}_k$, the state reads $\texttt{F}_1, \ldots, \texttt{F}_p$, and the state updates $\texttt{G}_1, \ldots, \texttt{G}_q$ that `Y` depends on. This footprint is independent of any parameters of `Y`. The lemma then states that `n` commutes with `Y` if it commutes with $\texttt{Z}_i$, `ReadState` $\texttt{F}_i$, and `UpdateState` $\texttt{G}_i$ for the appropriate indices `i`. We generate these statements with a Python script (see Section 5.2.1), and prove them with a custom proof tactic. We first expand the definition of `Y`, which results in a proof goal `Commute m n` where `m` is a monadic expression. Our proof tactic is then recursively defined on the structure of `m`: if `m` is composed of other monadic functions, for example with `Bind`, `ForLoop`, or a conditional statement, our tactic checks whether the components commute with `n`. Otherwise, `m` equals by construction $\texttt{Z}_i$, `ReadState` $\texttt{F}_i$, or `UpdateState` $\texttt{G}_i$ for an index `i`, and our proof tactic applies the corresponding assumption.

To be able to prove that two auxiliary functions `Y` and `Z` commute, we develop another automated proof tactic. With a proof goal `Commute m n` where both `m` and `n` are monadic expressions, the tactic is defined recursively on both `m` and `n`. We explain the most interesting cases below:

- If `m` has the form `ReadState f` and `n` has the form `UpdateState g`, or the other way around, our tactic uses Isabelle's *simp* tactic to try to prove that `f (g s) = f s` for all states `s`.
- If `m` has the form `UpdateState g` and `n` has the form `UpdateState g'`, our tactic uses *simp* to try to prove that `g (g' s) = g' (g s)` for all states `s`.
- If `m` is an auxiliary function for which we proved a commutativity lemma, our tactic applies that lemma, and recursively tries to prove the assumptions of the lemma. The case that `n` is an auxiliary function is similar.
- If `m` is composed of the monadic functions $\texttt{m}_1, \ldots, \texttt{m}_k$, for example with `Bind`, `ForLoop`, or a conditional statement, our tactic recursively tries to prove that $\texttt{m}_1, \ldots, \texttt{m}_k$ commute with `n`. The case that `n` is composed of other monadic functions is similar.

So effectively, if `Y` transitively depends on the state field reads and updates $\texttt{F}_1, \ldots, \texttt{F}_m$ and `Z` transitively depends on $\texttt{G}_1, \ldots, \texttt{G}_n$, our tactic checks whether each combination of $\texttt{F}_i$ and $\texttt{G}_j$ commutes.

While commutativity allows us to move away functions that block simplification lemmas, it is unclear how to adapt our Hoare proof tactic to do this automatically. In the example above we commuted `WriteGPCapReg` and `WriteData` in one direction, but in other proofs we may need to commute them in the other direction. We even may need to commute them in different directions in the same proof. Unfortunately, we can configure the *simp* tactic in Step 3 to either rewrite `Y (Second (Z s))` to `Z (Second (Y s))`, or the other way around, but not both, as this would cause *simp* to loop.

5.1.5 Monadic Hoare logic

To overcome the problem described above, we define a variant of Hoare triples where pre- and postconditions are monadic functions. The monadic structure keeps distinct steps separated, which allows us to apply the commutativity tactic to the postcondition. Our new Hoare proof tactic does this automatically: it constructs a precondition of the form `Bind m' post`, where `m'` is obtained from `m` by removing all steps that commute with the postcondition and later steps. We illustrate this with some examples, and define monadic Hoare triples and the new tactic in more detail later in this subsection.

We first return to the example in Section 5.1.3 where our original Hoare proof tactic gets stuck. There we considered the postcondition `MemCap s a = cap` and the following expression as `m`:

```
1 do {
2   x ← WritePCC p;
3   x ← WriteGPCapReg v;
4   WriteData w
5 }
```

To use the postcondition with our new proof tactic, we rewrite it in a monadic style:

```
do {
  read ← ReadState (λs. MemCap s a);
  Return (read = cap)
}
```

Our new Hoare proof tactic traverses `m` backwards and tries to commute each step of `m` with the postcondition. Because `WriteData w` does not commute with the postcondition, the tactic simply prepends it to the postcondition, and uses the result as the intermediate postcondition for the next iteration:

```
do {
  x ← WriteData w;
  read ← ReadState (λs. MemCap s a);
  Return (read = cap)
}
```

Because `WriteGPCapReg v` commutes with the intermediate postcondition above, it does not affect its value. The tactic therefore uses the same postcondition as the postcondition for the next iteration. The same holds for `WritePCC p`. As `m` has been traversed, the tactic outputs the intermediate postcondition above as the constructed precondition. Note that the new tactic does not use any simplification lemmas, thus avoiding the situation where our previous Hoare proof tactic gets stuck.

For our next example, suppose we would like to prove that `SignalException t` sets the EPC to a value that satisfies some condition `f`. We construct the following postcondition:

```
do {
  epc ← ReadState EPC;
  Return (f epc)
}
```

We unfold the definition of `SignalException` (see Appendix A.2 on pages 164–168) and run our new Hoare proof tactic, which constructs the precondition shown in Figure 5.1 on the following page. Here, the tactic removed all the steps from `SignalException` that do not affect the postcondition. Moreover, it removed all steps that change the state, which makes it feasible to expand the definition of the state monad without causing a combinatorial explosion. Doing so gives the following (non-monadic) precondition:

```
if ExceptionLevel s
then f (EPC s)
else if IsSome (BranchDelay s) or IsSome (BranchDelayPCC s)
     then f (PC s - 4)
     else f (PC s)
```

This condition is easy to work with, both for manual proofs and for Isabelle's proof tactics, especially when compared to `SignalException`'s 227 line definition.

We formally define monadic Hoare triples below and then describe the monadic Hoare proof tactic in more detail.

Definition 5.8. Let `pre` be a monadic condition of type `State ⇒ Bool × State`, `m` a monadic function of type `State ⇒ 'a × State`, and `post` a monadic condition of type `'a ⇒ State ⇒ Bool × State`. Then the Hoare triple `pre`, `m`, `post` holds if for all `s` for which the value returned by `pre` in `s` is true, the value returned by `Bind m post` in `s` is

```
do {
  exl ← ReadState ExceptionLevel;
  if not exl
  then do {
         branch ← ReadState BranchDelay;
         if IsSome branch
         then do {
                pc ← ReadState PC;
                Return (f (pc - 4))
              }
         else do {
                branch ← ReadState BranchDelayPCC;
                if IsSome branch
                then do {
                       pc ← ReadState PC;
                       Return (f (pc - 4))
                     }
                else do {
                       pc ← ReadState PC;
                       Return (f pc)
                     }
              }
       }
  else do {
         epc ← ReadState EPC;
         Return (f epc)
       }
}
```

Figure 5.1: The precondition that our Hoare proof tactic constructs when run on `SignalException t` and the postcondition that reads the EPC and then requires that the condition `f` holds for the EPC. The tactic removed all the steps from `SignalException t` that do not affect the postcondition. By construction of our proof tactic, the shown precondition, `SignalException t`, and the mentioned postcondition form a Hoare triple.

also true. Formally:

```
HoareTriple pre m post ≡
  for all s.
  if First (pre s)
  then First (Bind m post s)
```

Note that we ignore the state that the pre- and postcondition produce in the definition above. To help create monadic conditions, we lift logical operators such as "=", `not`, `or`, and `and` to the state monad. For example, the lifted version of `and` takes two arguments `m` and `n` of type `State ⇒ Bool × State` and returns the monadic function of the same type that, given a state `s`, returns the value `First (m s) and First (n s)` paired with `s`.

We define a small generalisation of the commutativity tactic to deal with the following special case. Consider the monadic function `UpdateState (λs. s(PCC := cap))` and suppose the first step of the postcondition is `ReadState PCC`. They do not commute, but we can change `ReadState PCC` into `Return cap`, because reading the PCC after it has been set to `cap` just returns `cap`, and then they do commute. The generalised tactic, which we call the swapping tactic, allows that: given two monadic expressions `m` and `n`, the tactic tries to find an `n'` such that first applying `m` and then `n` is the same as first applying `n'` and then `m`. The tactic is defined recursively on `n`. The most interesting cases are the base cases: if `n` has the form `ReadState f`, the tactic checks whether there is a simplification lemma `f (Second (m s)) = c`. If so, changing `n` into `Return c` allows it to be swapped with `m`. If there is no simplification lemma, or if `n` is one of the other bases cases such as an auxiliary function or `UpdateState g`, the tactic checks whether `m` commutes with `n` by invoking the commutativity tactic. If they commute, `n` does not need to change in order to swap with `m`.

We define the new Hoare proof tactic in a similar way to the previous Hoare tactic: given a monadic function `m` and a monadic postcondition `post`, the new tactic constructs a monadic precondition `pre` such that `pre`, `m`, `post` is a Hoare triple. It is defined recursively on the structure of `m`:

1. If `m` has the form `Return x`, the tactic returns `post x`.
2. If `m` has the form `ReadState f`, the tactic simply prepends `m` to the postcondition, resulting in `Bind (ReadState f) post`. Invoking the swapping tactic has no use here: if `post` depends on the value that `m` returns, we cannot swap them, and if `post` does not depend on the value, we simplify the constructed precondition with the simplification lemma `Bind (ReadState f') (λ_. n) = n` in a later step.
3. For the case that `m` has the form `UpdateState g`, recall that `UpdateState g` returns `()`, which is the value of type `Unit`. Our tactic invokes the swapping tactic on `m` and `post ()`. If they can be swapped by changing `post ()` into `post'`, the

tactic returns `post'`. Otherwise, it prepends `m` to the postcondition, resulting in `Bind (UpdateState g) post`.

4. The case that `m` is an auxiliary function combines aspects of the previous two cases. First, the tactic checks whether `m` can be swapped with `post` if we disregard the value that `m` produces. More precisely, it introduces a new variable `x` and invokes the swapping tactic on `m` and `post x`. If they cannot be swapped, the tactic prepends `m` to the postcondition, resulting in `Bind m post`. If they can be swapped by changing `post x` to `post' x`, the tactic returns `Bind (ReadState (ValuePart m)) post'`. Here, `ValuePart` only returns the value that `m` produces and `ReadState` turns it into a monadic expression again, making it explicit that the state that `m` produces is not relevant. If `post'` does not depend on the value that `m` returns, we can remove `ReadState (ValuePart m)` with the simplification lemma stated in Step 2.
5. If `m` has the form `Bind m' n'`, the tactic traverses `m` backwards. First, it recursively constructs a precondition of `n'` and `post`, which we call `intermediate`. It then invokes *simp* to obtain a simpler, but equivalent function `intermediate'`. In contrast with our previous Hoare proof tactic, we do not supply *simp* with any lemmas about specific auxiliary functions. Finally, the tactic recursively constructs a precondition of `m'` and `intermediate'`, which it uses as the precondition of `Bind m' n'` and `post`.
6. If `m` is a conditional statement, such as an if-statement or a case split, the tactic recursively constructs preconditions of their components and combines them with the same conditional statement. For example, if `m` equals `if b then n else n'`, and the constructed preconditions of `n` and `n'` are respectively `pre` and `pre'`, then the constructed precondition is `if b then pre else pre'`.
7. If `m` has the form `ForEachLoop l m'` or `ForLoop i j m'`, our tactic tries to prove that `post ()` is an invariant of the loop. First, it recursively constructs a precondition `pre` of `m'` and `post`. It then invokes *auto* to try to prove that `post ()` implies `pre`. If it does, then the tactic returns `post ()` as the constructed precondition. Otherwise, the tactic prepends the entire loop to the postcondition. We can manually supply a loop invariant to be used instead of this behaviour.

The cases for `ExtendState` and `TrimState` are missing above because we initially thought that our tactic could not be made to work with them. In hindsight, we believe it could be feasible to adapt our proof tactic, but we already found another solution (see Section 5.2.2).

5.2 Proof overview

With our automated proof tactics established above, we now give an overview of our proof development. It consists of two parts: the first is the proof of Theorem 3.17 on page 111, which states that any execution step allowed by CHERI-MIPS is also allowed by our abstraction (see Chapter 3), and the second consists of the proofs of our compartmentalisation theorems above that abstraction (see Chapter 4). Because Theorem 3.17 directly refers to the CHERI-MIPS semantics, all the challenges about the size, details, and changing nature of the specification that we discussed in Section 1.5.2 apply here. The compartmentalisation theorems, on the other hand, only refer to the CHERI-MIPS semantics through our abstraction, and since our abstraction is a small artefact that remains relatively stable, those challenges do not apply here. Because the proofs in the first part are considerably more involved than those in the second part, we focus this section on the former.

Our automated proof tactics (see Section 5.1) make the proof of Theorem 3.17 resilient to small changes in the CHERI-MIPS architecture, such as the change that jump instructions no longer raise an exception when used on local capabilities. Larger changes, such as the addition of a new instruction, may require changes to the proof, such as adding lemmas about the new instruction. To minimise the effort needed, we use Python scripts to generate parts of the Isabelle source files that comprise the proof (see Section 5.2.1).

The structure of the proof of Theorem 3.17 is as follows. We first define a simpler, but equivalent, alternative for each auxiliary function in the CHERI-MIPS specification (see Section 5.2.2). We then define generally applicable lemmas about the CHERI-MIPS specification, including lemmas about address translation (see Section 5.2.3). Finally, we show that CHERI-MIPS satisfies all the security properties of our abstraction (see Section 5.2.4). We translate each property to a Hoare triple, use a Python script to generate lemmas and proofs for repetitive cases, and manually state and prove the conceptually interesting ones. In Table 5.1 on the following page we describe the sizes of these parts.

5.2.1 Python scripts

Our Python scripts output Isabelle source code that, once generated, does not depend on the scripts any more: Isabelle does not interact with our Python scripts, and verifies the source files as if we had written them by hand. This means that our scripts do not need to be trusted. Our approach is less powerful than using Isabelle's ML interface, which can inspect Isabelle's internal state, but it is easier to debug and needs less maintenance when upgrading the version of Isabelle.

The input of our Python scripts is a dependency graph of the auxiliary functions in the CHERI-MIPS specification, describing which state fields an auxiliary function reads,

	Manual lines	Generated lines
Machine word lemmas	1,672	0
Lemmas about L3 library	760	0
Simpler, but equivalent definitions	1,108	3,292
Automated proof tactics	1,258	867
Lemmas about CHERI-MIPS	3,010	396
Abstraction	695	0
Connecting CHERI-MIPS to the abstraction	2,209	0
Proof of `IsUnpredictable` invariant	21	896
Proof of `ExceptionSignalled` invariant	25	1,180
Proof of `StateIsValid` invariant	87	1,176
Proof of Loading data	687	0
Proof of Storing data	642	0
Proof of Executing instructions	193	0
Proof of Loading capabilities	248	0
Proof of Storing (local) capabilities	268	0
Proof of Memory invariant	670	1,221
Proof of Restricting capabilities	733	1,582
Proof of Sealing capabilities	124	0
Proof of Unsealing capabilities	119	0
Proof of Special capability register access	105	0
Proof of Address translation	155	832
Proof of Capability invariant	648	1,218
Proof of Invoking capabilities	189	0
Proof of Raising exceptions	260	1,288
Proof of Theorem 3.17	46	0
Lemmas about compartment authorities	726	0
Compartmentalisation theorems	1,598	0
Total	**18,256**	**13,948**

Table 5.1: The number of non-comment, non-white space lines of Isabelle/HOL in our proof development. The generated lines are generated with Python scripts (see Section 5.2.1). For comparison, the L3 specification and its Isabelle/HOL export are respectively 7k and 59k non-comment, non-white space lines. The increase in size is caused by several factors: the export defines a shallow embedding of L3's behaviour in HOL, it uses a separate monadic step for each calculation (see Section 5.2.2), and it uses Isabelle's ML interface to create definitions.

which state fields it changes, and which other auxiliary functions it calls. We maintain the dependency graph manually, but it could be automatically generated from the L3 source. Each script scans our Isabelle source files for a particular placeholder comment, generates Isabelle code based on the dependency graph, and replaces the comment with its output. If a change in the CHERI-MIPS specification does not directly affect our security properties, for example, if a new instruction is added that does not manipulate capabilities or access memory, we typically only need to update the dependency graph, run our Python scripts, and our proof is valid again.

As described in Section 5.1.4, for each low-level auxiliary function `Y` we generate a lemma stating that an arbitrary function `n` commutes with `Y` if it commutes with `Y`'s dependencies. In the subsections below we give more details about our other scripts.

5.2.2 Simpler, but equivalent definitions

To make the auxiliary functions in the specification more readable and suitable for our automated proof tactics, we state a simpler, but equivalent definition for each of them. We do not state the alternative definitions by hand, but instead we apply *simp*, Isabelle's term rewriting tactic, to each auxiliary function, and whatever the auxiliary function is rewritten to becomes the alternative definition. We achieve this using *schematic lemmas* in Isabelle: for each auxiliary function `Y`, we state the schematic lemma `Y = ?x`, and prove it by unfolding the definition of `Y` and applying *simp* with the simplification lemmas that we describe later in this section. Isabelle then instantiates the schematic variable `?x` with the rewritten definition to make the proof succeed. We generate the schematic lemmas and their proofs with a Python script, so the only manual effort needed is stating and proving the simplification lemmas.

The first kind of simplification lemma makes the auxiliary functions less verbose. The main cause of verbosity is the fact that the Isabelle export of an L3 definition uses a separate imperative step for each calculation. For example, in the following excerpt from the definition of `LoadCap` (see Figure 2.17 on page 78) the application of `BaseAddress`, `ExtractWord`, and the equality are distinct steps (respectively Line 4, Line 5, and Line 7):

```
1  do {
2    b ← do {
3        v ← do {
4            v ← Return BaseAddress v;
5            Return ExtractWord 36 2 v
6          };
7        Return (a = v)
8      };
9    if b then ... else ...
10 }
```

With the simplification lemma that `Bind (Return x) m = m x` for all `m` and `x`, we can rewrite the entire excerpt to one line:

```
if a = ExtractWord 36 2 (BaseAddress v) then ... else ...
```

We prove other simplification lemmas that rewrite L3 library functions to native Isabelle functions, that remove unnecessary machine word manipulations, such as repeated casts, and that remove unnecessary monadic steps, such as `ReadState f` if its resulting value is not used or if it is a duplicate read.

The second kind of simplification lemma makes the auxiliary functions suitable for our commutativity tactic (see Section 5.1.4). A common idiom in the L3 specification is to read the `ProcessorState`, change a subfield, and update the state with the result. For example, the following is the idiomatic way to change the PC:

```
1 do {
2   v ← ReadState ProcessorState;
3   v' ← Return (v(_PC := pc));
4   UpdateState (λs. s(ProcessorState := v'))
5 }
```

Our commutativity proof tactic considers each step separately, and therefore treats the above snippets as if its footprint is the entire `ProcessorState`. Although the snippet commutes with any other processor state update that leaves the PC unchanged, our tactic cannot prove this. To avoid this problem, we prove a simplification lemma that shows the above is equivalent to:

```
UpdateState (λs. s(ProcessorState := (ProcessorState s)(_PC := pc)))
```

We prove similar lemmas for the other subfield updates of `ProcessorState`.

The third kind of simplification lemma removes `ExtendState` and `TrimState` from the auxiliary functions, ensuring that the monad always uses the CHERI-MIPS machine state as its state. Recall that `ExtendState` and `TrimState` are used to implement L3's mutable variables (see Section 2.4.6). For example, the following made-up L3 snippet reads the capability at an address `a`, initialises the mutable variable `cap` with the result, changes the variable by removing the tag of the capability, and writes the result back to address `a`:

```
1 var cap = ReadCap(a);
2 cap <- setTag(cap, false);
3 WriteCap(a, cap);
```

In its Isabelle export below, the capability is read from address `a` (Line 2), and then the state of the monad is extended to `Capability × State` to store the result (Line 3). The capability is then read from the extended state (Line 4), its tag is removed (Line 5), and it is written back to the state (Line 6). Finally, the capability is read from the state again (Line 7), and written to address `a` (Line 8). Here, the state of the monad is first trimmed back to the type `State` before invoking `WriteCap`.

```
1  do {
2    v1 ← ReadCap a;
3    ExtendState v1 (do {
4                      v2 ← ReadState First;
5                      v3 ← Return (v2(Tag := False));
6                      v4 ← UpdateState (λs. (v3, Second s));
7                      v5 ← ReadState First;
8                      TrimState WriteCap (a, v5)
9                    })
10 }
```

Because our automated proof tactics do not support extended states, we explicitly pass around the values of mutable variables instead of storing them in an extended state. For example, we rewrite the Isabelle export above to the version below, where each return value is a pair of the original return value and the value of the mutable variable. Steps that do not change the mutable variable simply pass it along in the second component (Lines 4, 5 and 7), and steps that change the variable return the new value (Lines 3 and 6). Steps that read the mutable variable copy the second component to the first (Lines 4 and 7).

```
1  do {
2    v1 ← ReadCap a;
3    (x, cap1) ← Return ((), v1);
4    (v2, cap2) ← Return (cap1, cap1);
5    (v3, cap3) ← Return (v2(Tag := False), cap2);
6    (v4, cap4) ← Return ((), v3);
7    (v5, cap5) ← Return (cap4, cap4);
8    WriteCap (a, v5)
9  }
```

An additional benefit of our transformation is that our verbosity-reducing simplification lemmas apply: we can rewrite the above to the following.

```
do {
  v1 ← ReadCap a;
  WriteCap (a, v1(Tag := False))
}
```

To make our transformation work for `ForEachLoop`, we define a new kind of loop that passes a value between iterations, which we use the store the values of mutable variables. Our new loop takes a list of type `'a List`, an initial value of type `'b`, and a monadic function of type `'a ⇒ 'b ⇒ State ⇒ 'b × State`. The first iteration is passed the initial value of type `'b`, the others are passed the return value of the previous iteration, and the loop itself returns the return value of the last iteration. We treat `ForLoop` as a special case of `ForEachLoop` that iterates over a list of consecutive integers.

As an example of applying all these simplification lemmas, we show the alternative definition of `LoadCap` in Figure 5.2 on the following page.

5.2.3 General lemmas

As part of our proof, we state several generally applicable lemmas. Some of these are conceptually interesting and would also be included in a pen-and-paper proof. For example, we prove that addresses are translated per page. Some are less interesting, but non-trivial results which one could skip over in a pen-and-paper proof, but which are necessary to get the details right. For example, many of our machine word lemmas fall in this category. Others are trivial results that we make available to Isabelle's proof tactics. Proof tactics do not expand definitions automatically and therefore miss obvious facts about them, such as the fact that `Tag NullCap = False`, so we prove this as a lemma. Below we give more detailed examples about the first two categories.

We first consider our machine word lemmas. Among other things, we define a bitwise order over words, where `x` $\leq$ `y` if each bit that is set in `x` is also set in `y`, and we show that it forms a complete boolean algebra with the standard operators such as `AND`, `OR`, and `NOT`. We often prove equality of two words by proving that each of their bits are equal, and to support this approach we prove lemmas that describe the `n`-th bit of an operation, for example, describing the `n`-th bit of a concatenation in terms of the bits of its operands. We also prove lemmas that describe the result of consecutive operations, such as combinations of (un)signed casts, slices, concatenations, shifts, additions, and restrictions to the lower or upper bits of a word. Finally, we prove that certain additions cannot overflow, such as the addition of the `n` lower bits of `x` to the all-but-`n` upper bits of `y`.

We also prove lemmas about the L3 library, which contains the definition of the state monad and new machine word operations, such as `ExtractWord` and `ExtractByte`. The for-loop of the state monad is defined with an awkward recursion, namely on the absolute value of the difference of two of its operands, so we show that it is a special case of the for-each-loop, which is defined using recursion on lists. Furthermore, we lift logical operators such as "`=`", `not`, `or`, and `and` to the state monad, as described in Section 5.1.5. We also prove lemmas describing the `n`-th bit of the new machine word operations, and lemmas about applying combinations of these operations and native Isabelle machine word operations.

We then define general concepts and prove general lemmas about CHERI-MIPS. For example, we define when a state is valid (see Section 2.5.1) and we define the semantics of unpredictable behaviour (see Section 2.5.4). We define our order over capabilities (see Section 3.2) and prove that it is reflexive and transitive, forming a pre-order. We also prove lemmas about operations that make the capability smaller or equal in this order, for

```
LoadCap (vAddr, link) ≡
  do {
    (pAddr, cca, a) ← AddressTranslation (vAddr, Load);
    b ← ReadState ExceptionSignalled;
    if b
    then do {
           v ← NextUnknown "capability";
           Return (Undefined v)
         }
    else do {
           v ← ReadState JTAG_UART;
           _1 ←
             if Slice 5 pAddr AND Mask 35 = Slice 2 (BaseAddress v) AND Mask 35
             then RaiseL3Exception (Unpredictable "Capability load attempted on UART")
             else do {
                    v ← ReadState TotalCore;
                    ForEachLoop (Sequence 0 (v - Suc 0))
                     (λcore. do {
                           v ← ReadState BaseAddressPIC;
                           if Slice 2 (v (IntToWord (NatToInt core))) AND Mask 35
                              ≤ Slice 5 pAddr AND Mask 35
                              and Slice 5 pAddr AND Mask 35
                                  < Slice 2 (v (IntToWord (NatToInt core)) + 1072) AND Mask 35
                           then RaiseL3Exception (Unpredictable "Capability load attempted on PIC")
                           else Return ()
                         })
                  };
           _4 ←
             if link
             then do {
                    _2 ←
                      if cca = 2
                      then RaiseL3Exception (Unpredictable "load linked on uncached address")
                      else Return ();
                    _3 ← UpdateState (SetLoadLinkFlag (Some True));
                    UpdateState (SetLoadLinkAddress (UnsignedCast pAddr))
                  }
             else UpdateState (SetLoadLinkFlag None);
           v ← ReadCap (Slice 5 pAddr AND Mask 35);
           aa ← ReadState MemAccessStats;
           aa ← UpdateState (λs. s(MemAccessStats := aa(BytesRead := BytesRead aa + 32)));
           Return if Second a
                  then v(Tag := False)
                  else v
         }
  }
```

Figure 5.2: The alternative definition of `LoadCap` (see Figure 2.17 on pages 78 and 79 for the original definition). Both definitions are equivalent, but this alternative definition is less verbose, replaces some L3 library functions such as `ExtractWord` with native Isabelle functions such as `Slice`, and fixes the type of the state monad to the CHERI-MIPS machine state `State` instead of using a dynamic state.

example, describing when `cap(Perms := p)` $\leq$ `cap` holds as a condition on `p` and `cap`. We define capability locations and the functions `CapReg`, `MemCap`, and `Cap` to enable addressing capabilities in a uniform way (see Section 3.3) and we prove commutativity lemmas about these functions. We use a Python script to generate lemmas describing that certain auxiliary functions in the specification do not alter the state. For example, `ReadGPCapReg` does not, which is captured by the lemma: `for all n s. Second (ReadGPCapReg n s) = s`. We prove that `AddressTranslation`, on which `TranslateAddr` is based, only changes the state if it causes unpredictable behaviour or raises an exception. As a final example, we show that virtual addresses are translated per page, which is captured by the combination of the following three lemmas. The first states that if two virtual addresses lie in the same page and one of them can be translated, then the other can also be translated:

Lemma 5.9. If the virtual addresses `vAddr` and `vAddr'` lie in the same page, that is, all but their lowest 12 bits agree (Line 2), and if `vAddr` can be translated with access type `t` in state `s` (Line 3), then `vAddr'` can also be translated with `t` in `s` (Line 4). Recall that `TranslateAddr` returns `None` if the translation fails (see Section 2.5.3).

```
1 for all vAddr vAddr' t s.
2 if vAddr AND NOT Mask 12 = vAddr' AND NOT Mask 12
3    and IsSome (TranslateAddr (vAddr, t) s)
4 then IsSome (TranslateAddr (vAddr', t) s)
```

The second lemma states that if two virtual addresses lie in the same virtual page and can be translated, then their translations lie in the same physical page:

Lemma 5.10. If the virtual addresses `vAddr` and `vAddr'` lie in the same virtual page, that is, all but their lowest 12 bits agree (Line 2), and if `pAddr` and `pAddr'` are the translations of respectively `vAddr` and `vAddr'` with access type `t` in state `s` (Lines 3–4), then `pAddr` and `pAddr'` lie in the same physical page, that is, all but their lowest 12 bits agree (Line 5).

```
1 for all vAddr vAddr' pAddr pAddr' t s.
2 if vAddr AND NOT Mask 12 = vAddr' AND NOT Mask 12
3    and TranslateAddr (vAddr, t) s = Some pAddr
4    and TranslateAddr (vAddr', t) s = Some pAddr'
5 then pAddr AND NOT Mask 12 = pAddr' AND NOT Mask 12
```

The third lemma states that if a virtual address can be translated, then the lowest 12 bits of its translation equal the lowest 12 bits of the virtual address:

Lemma 5.11. If `pAddr` is the translation of the virtual address `vAddr` with access type `t` in state `s` (Line 2), then the lowest 12 bits of `pAddr` equal the lowest 12 bits of `vAddr` (Line 3).

```
1 for all vAddr t pAddr s.
2 if TranslateAddr (vAddr, t) s = Some pAddr
3 then pAddr AND Mask 12 = UnsignedCast vAddr AND Mask 12
```

We prove two invariants about ghost state: we prove that auxiliary functions do not clear the `IsUnpredictable` and the `ExceptionSignalled` flags. In other words, if an auxiliary function caused unpredictable behaviour or raised an exception, auxiliary functions that are called later do not undo this.

Finally, we prove general lemmas about compartment authorities (see Section 4.3), which we use in the proofs of our compartmentalisation theorems. We define an order over compartment authorities by lifting the orders of its fields (which is boolean implication for `SystemRegisterAccess` and set inclusion for the others). We also lift standard operators such as union and big union, and we prove that this forms a complete boolean algebra. We connect the order over capabilities with the order over compartment authorities with the following lemma:

```
for all cap cap'.
if cap ≤ cap'
then GetAuthority cap ≤ GetAuthority cap'
```

5.2.4 Proving security properties

To prove the top-level theorem of Chapter 3, which states that any execution step allowed by CHERI-MIPS is also allowed by our abstraction (see Theorem 3.17), we consider a labelled execution step `(s, label, s')` in CHERI-MIPS, and prove that it satisfies all the security properties of our abstraction. Below we describe our experience of proving these.

Some of the requirements of our security properties directly correspond to assertions in CHERI-MIPS instructions, which makes them relatively easy to verify. For example, our properties about memory accesses and capability manipulations require that the capability that is used as authority is valid, unsealed, and has the corresponding permission. Our proof verifies that the relevant instructions indeed check these conditions.

Other requirements follow from a combination of assertions in instructions and deeper properties about CHERI-MIPS. For example, our memory access properties require that the physical footprint of the access lies within the translated memory region of the capability that is used as authority. Instructions that access memory, on the other hand, check whether the virtual address of the access is greater than or equal to the base of the capability, and whether the 65-bit addition of the address and the length of the access is smaller than or equal to the 65-bit addition of the base and the length of the capability. Furthermore, the aligned instructions check whether the virtual address is aligned, whereas left-unaligned instructions reduce the access length to stop at an alignment boundary, and right-unaligned instructions truncate the virtual address to that alignment boundary and reduce the access length to the remaining length. To prove that these checks imply our requirement, we need many of our machine word lemmas and our result that addresses are

translated per page. It is unsurprising that we found several bugs in the bounds checks of memory access instructions (see Chapter 6).

The semantics of CHERI-MIPS is non-deterministic, but this does not complicate our proofs. In CHERI-MIPS, unpredictable execution steps cannot access memory, change capabilities, or change address translations (see Section 2.5.4), so our memory, capability, and address translation invariants hold by construction. Most of our other security properties assume that a memory access or capability manipulation occurs during the execution step, and since this assumption is not met, these properties hold trivially. The only non-trivial result we prove about unpredictable execution steps is Property 3.4 (Executing instructions). This property assumes that the execution step does not raise an exception, which is true for unpredictable execution steps, and then requires that the PCC is valid, unsealed, has the PermitExecute permission, and contains the address of the next instruction in its memory region. CHERI-MIPS performs these checks just before instruction fetch. Our proof verifies that no unpredictable behaviour is caused before these checks, and if one of them fails, no unpredictable behaviour is caused afterwards either.

The imperative style of the specification means that our security properties become gradually true and our invariants can be temporarily broken during an execution step, leading to lots of bookkeeping. Recall that our proofs follow this imperative style because it is infeasible to expand the state monad all at once (see Section 5.1.1). We transform each security property to a monadic Hoare triple (see Sections 5.1.3 and 5.1.5) and work backwards through the specification of an execution step, transforming the post condition whenever necessary. For example, some instructions change capabilities in a loop, so our intermediate post conditions need to keep track of where we are in the loop. Another example is that permission checks are spread over different auxiliary functions, so we need to keep track of what has been checked already. A final example are branch instructions: they first change `BranchToPCC`, and at the end of the instruction they copy that to `BranchDelayPCC`. If our post condition refers to a capability at a location `loc` or a register `r`, at some point in the proof our intermediate post condition needs to substitute the location `RegBranchDelayPCC` for `RegBranchToPCC`.

The size of the proof of a property is related to how many instructions are relevant, but has little relation to the difficulty of the proof. For example, Property 3.5 (Loading capabilities) assumes that a capability is loaded during the execution step, and only two instructions load capabilities, namely CLC and CLLC. The proof is small but intricate, as we saw in our discussion about bounds checks. To prove our address translation invariant, on the other hand, we need to check every instruction, leading to a large proof, but the proof is simple: for the six instructions that change address translations we check that they require the PermitAccessSystemRegisters permission, and for the other instructions we check that they commute with `TranslateAddr`.

Our Python scripts turned out to be a versatile proof development tool. They make it easy to experiment with invariants: by generating a lemma and proof for each auxiliary function, letting Isabelle verify these proofs, and inspecting the failed ones (if any), one can quickly make an educated guess whether the invariant is true. We used this approach iteratively, either refining the invariant, improving the generated proofs, or manually proving difficult cases. They also reduce the effort needed to update our proofs when the CHERI-MIPS specification changes, as described in Section 5.2.1. Besides our invariants, we use Python scripts for security properties where a significant number of instructions are relevant, namely our property about raising exceptions, and, since almost all instructions can complete a pending branch, our property about restricting capabilities.

Chapter 6

Bugs found by proof work

The goal of our proofs is to provide confidence that our security properties hold, not just to find bugs, but we did find some bugs, both in the L3 specification and the prose specification. CHERI-MIPS was already reasonably mature when we started our proofs, so these bugs are not very numerous, but each of the found bugs could lead to a security vulnerability. It is instructive to see what bugs can remain, even in a carefully considered and reviewed design, without proof.

The most severe bug we found allowed any valid, unsealed capability to be used to load data, even if it did not have any permissions at all. The bug was introduced with the following change: "The CLC (capability load capability) and CLLC (capability load-linked conditional) instructions will now strip loaded tags, rather than throwing an exception, if the Permit Load Capability permission is not present" [162, §1.5]. Not requiring this permission is fine if the tag is stripped, because invalid capabilities cannot be used as authority. However, not requiring the PermitLoad permission in that case instead is a bug: even if the tag is stripped, one can inspect the byte representation of the loaded capability and copy it to GPRs. Version 6 of the ISA contained this bug, both in the L3 specification and the prose specification, and it was fixed by always requiring the PermitLoad permission, even if the PermitLoadCapability permission is present.

We uncovered three bugs in the bounds checks of memory access instructions. The addition in the access length check could wrap, allowing all memory access instructions to access the end of the address space without authority. This was fixed by performing the addition in 65-bit instead of 64-bit, preventing wrapping. The access length check of legacy instructions contained a second bug, allowing them to access one byte past the region of memory the DDC has authority to. Finally, the right-unaligned load instructions used the wrong virtual address in the bounds checks, allowing unauthorised loads. All three bugs were present in the L3 specification, and the third bug was also present in the Bluespec hardware implementation. The prose specification omits the specification of legacy instructions, including the unaligned accesses, and for that reason does not

contain the last two bugs. Because it specifies bounds checks using mathematical integers, wrapping is not an issue, and therefore it does not contain the first bug either.

In version 6 of the ISA, the effects of capability invocation using CCallFast were delayed by one instruction, just like branch instructions (see Section 2.1.3). In particular, the instruction in the delay slot was still part of the protection domain that executed CCallFast. We found a bug here: CCallFast placed the unsealed code capability in the PCC, which means the instruction in the delay slot could access it, while CCallFast should have placed it in the branch delay PCC. This bug broke compartment isolation, as it allowed a compartment to transition to another compartment and steal its unsealed code capability just before the domain transition actually took place. This bug was present in both the L3 and the prose specification. We also found counter-intuitive behaviour here: the unsealed data capability could be accessed by all instructions after the delay slot, regardless of whether the domain transition would succeed. This led to the discovery of a vulnerability in CheriBSD: raising an exception in the CCallFast delay slot gave the exception handler access to the unsealed data capability. By registering a signal handler to deal with segfaults and triggering a segfault in the delay slot, the signal handler could obtain the unsealed data capability of another protection domain and use it to access memory. One could conceivably fix this in CheriBSD, but correct code would be harder to write and understand, so the designers of CHERI removed the CCallFast delay slot altogether.

We used our proofs as a regression "test" whenever new instructions were added to CHERI-MIPS. During the development of the CBuildCap instruction in the L3 specification, our property about restricting capabilities failed, uncovering that CBuildCap created a capability with the wrong base.

At first, some of our security properties assumed that the processor mode was not kernel mode, which meant they did not apply to compartmentalised kernel code. To change this, we replaced this assumption with the assumption that the PCC does not have the PermitAccessSystemRegisters permission. When rerunning our proofs, our property about special capability register access failed, uncovering that exception return (ERET) could access the EPCC without that permission. This bug was present in the L3 specification.

Chapter 7

Related work

There is a long history of formal verification, going back to at least the 1940s, when Alan Turing proved the correctness of a program that computes `n!` through repeated addition [153, 97]. We restrict our discussion to the two areas that are most relevant to our work, namely formal security properties for architectures (see Section 7.1) and for capability systems (see Section 7.2). We separately discuss research that builds on our work (see Section 7.3).

7.1 Architectural security properties

In terms of proving formal properties for production-scale architectures, the closest related work is Reid's work within Arm. He formalised 59 properties of Armv8-M, based on the prose statements in the architecture document, and used an SMT model checking approach to verify that they hold [120]. These properties are stated in terms of an authoritative, formal model of Armv8-M, which Reid et al. created by shifting essentially the entire Armv8-M and Armv8-A sequential ISA specifications from pseudocode to machine-readable definitions [119, 122, 121]. These specifications are an order of magnitude larger, and much more complex, than CHERI-MIPS. Furthermore, Reid's SMT approach is largely automated, while our proofs require theorem-proving expertise. On the other hand, Reid's properties are much more specific than the properties we consider. For example, one of his properties states that on exception entry, the stack selection that was active before the exception should be recorded in bit two of the LR register. Other properties state that after entry to lockup, at least one bit in Fault Status Register CFSR should be set, the debug view of the program counter must equal `0xEFFFFFFE`, and HFSR.FORCED should not be modified. Although these properties are security-relevant, they in themselves do not capture security guarantees of Armv8-A, while our properties capture security guarantees that are strong enough to prove a use case correct (see Theorem 4.9 on page 124).

Bauereiss proved correctness of a purely functional characterisation of Armv8-A address translation [6]. His proof is based on the Sail translation of the authoritative Armv8-A model that we described above. We described Sail and this translation process in Section 1.4.1. Although Bauereiss's property in itself is not a security property, it is relevant when proving security properties about memory accesses. In later work, which we discuss in Section 7.3, Bauereiss uses the property to prove security properties for Morello. There is no direct counterpart of his property in our proofs: address translation in Armv8-A is done in hardware, while MIPS leaves all this to system software.

Schwarz and Dam developed a framework to prove information flow properties of architectures [134]. Given an initial labelling of state components as "secret" or "public", their framework automatically relabels secret components as public components until it can prove *non-interference*, which states that secret components cannot influence public components. They developed their framework in the interactive theorem prover HOL4 [59], and applied it to the HOL4 export of the Armv7 and MIPS-III models that Fox et al. developed in L3 [55, 53]. The MIPS-III model is the predecessor of the CHERI-MIPS specification that we use (see Section 2.3). Schwarz and Dam recognise that their framework cannot exclude information leaks through unpredictable behaviour [134, §7], stating the same reason we give in Sections 1.5.3 and 3.5.1. Leaving unpredictable behaviour aside, an information flow property such as "the memory contents at address `a` cannot influence general purpose registers if one does not possess a capability with authority to load from `a`" would be a good addition to the properties of our abstraction (see Section 3.5.1). However, Schwarz and Dam focus on information flow between registers, abstracting away from the concrete behaviour of the memory subsystem, such as address translation and memory protection, so their framework would not be able to prove such a result.

PUMP [36, 35] augments all registers and memory in the Alpha architecture [2] with uninterpreted metadata, which can be used to support multiple hardware security policies. These policies are defined by *rules* that determine whether an instruction step is allowed based on the metadata of the PC, the instruction, its operands, and, if applicable, the memory location that is stored to or loaded from, and on the instruction opcode. Rules also determine the metadata of the new PC, and, if applicable, the metadata of the result of the instruction. An instruction checks the PUMP cache to see whether an applicable rule has been computed already, and otherwise traps to the *miss handler*, which runs the policy's software, inserts the computed rule in the cache, and restarts the instruction. To support a wide range of policies, PUMP associates one 64-bit word of metadata to each 64-bit word of data. In particular, the metadata can hold a pointer to an arbitrary large data structure in memory. To accommodate the size of the metadata, PUMP extends the 64-bit registers of the base architecture to 128-bit, and doubles the capacity of the L2 cache. To avoid stalling the processor, the L1 cache is kept the same size, which reduces its effective capacity by half. The effective capacity of the main memory is also halved.

Two example policies are relevant to our work. The first adds memory safety to heap allocations by tagging each memory location with a *lock*: the ID of its allocation (if it is allocated), and a *key*: the ID of the allocation it points to (if it is a pointer). When dereferencing a pointer, the policy checks whether the key of the pointer matches the lock of the target. The second policy provides memory isolation between compartments. Each memory location is tagged with the ID of the compartment it belongs to, and the IDs of the compartments that can read, write, or jump to that location. During a memory access, the policy checks whether the compartment that the PC belongs to is allowed to perform the type of memory access according to the target's metadata. One can use a central monitor to dynamically create a hierarchy of compartments. De Amorim et al. [7] defined abstract specifications for several policies, including these two, proved by refinement that the policies are correct, and mechanised their proofs in Coq [14]. However, their proofs are based on an idealised PUMP architecture, which contains only ten instructions, and which models the machine state as only a PC, general purpose registers, and memory. The proofs do not cover system registers, privileged execution, virtual memory, exceptions, or unpredictable values.

Ferraiuolo et al. [48] describe HyperFlow, a RISC-V-based ISA and hardware implementation that prevents information leaks, even through timing channels. The architecture augments registers and memory pages with tags that hold security labels, describing the confidentiality and the integrity level of that location, and it adds instructions that manipulate these security labels, enabling applications to define flexible information flow policies. To prove that the HyperFlow hardware implementation indeed enforces timing-sensitive non-interference, Ferraiuolo et al. implement it in ChiselFlow, a security-typed hardware description language, which uses the SMT solver Z3 [98] to discharge proof obligations. Ferraiuolo et al. do not state or prove any architectural security guarantees for HyperFlow, but Zagieboylo et al. [174] do this for a similar architecture: they prove timing-sensitive non-interference modulo downgrading and non-malleability for programs executing on their ISA. The ISA is somewhat idealised, for example, it does not include exception configuration and handling, and their proofs are not mechanised.

Intel Software Guard Extensions (SGX) is an extension to the x86 ISA that provides integrity and confidentiality to programs running on a potentially malicious system [66, 24]. It introduces *enclaves*, protected execution contexts, and isolates enclave memory from the rest of the system, including privileged software such as the OS kernel or hypervisor. Sinha et al. [143] develop Moat, a system for statically verifying confidentiality properties of enclave programs. Moat uses a formal architecture model that Sinha et al. created by extending BAP [17] with the new SGX instructions, mapped into BoogiePL [9]. Their approach is largely automated: Moat invokes the Boogie program verifier [9], which in turn uses Z3 [98] to automatically verify the generated SMT conditions. However, Sinha

et al. do not discuss validation of their SGX model, or its relationship with the complex x86 system semantics.

Leaving security aside, there is extensive work on formal ISA modelling for hardware verification, for example for x86 in ACL2 [57, 70], in Coq [23], for RISC-V [169], and for Arm [122].

7.2 Security properties for capability systems

Turning to security properties for capability systems, we first discuss work that uses capability systems to enforce well-bracketed control flow and local state encapsulation. These are desirable properties that the CHERI C compiler (see Section 1.3.5) currently does not provide, because of performance reasons. Skorstengaard et al. [144] develop a calling convention that provides these properties (with the expected performance cost): using local capabilities, they limit the memory region where capabilities to the stack can be leaked, but they still need to clear the unused part of the stack between invocations. To avoid this, in later work they introduce *linear* capabilities [145], which are capabilities that cannot be duplicated. The crux here is that if one sends a linear capability and receives it back, it cannot have been leaked. Georges et al. [56] improve the first calling convention in a different way, introducing *uninitialised* capabilities, which give read/write access to memory without exposing its initial contents, removing the need to clear parts of the stack. The authors prove that their respective calling conventions enforce well-bracketed control flow and local state encapsulation, and a central part of these proofs is a property called *capability safety*. Both our compartmentalisation theorems (Theorems 4.4–4.7) and capability safety show that arbitrary code is limited to the set of capabilities it has access to, which allows reasoning about untrusted code. However, capability safety is a more sophisticated property that also allows reasoning about intermingled trusted and untrusted code. In particular, capability safety crosses domain transitions, while our theorems only apply up to, or up to and including a domain transition. On the other hand, capability safety is defined in terms of a step-indexed Kripke logical relation, which may make it difficult to understand for practitioners, while our theorems are defined in terms of simple traces. Furthermore, their results are based on idealised capability machines, inspired by CHERI but much simpler: they model the machine state as only a PC, general purpose registers, and memory, and do not cover system registers, privileged execution, virtual memory, exceptions, or unpredictable values, for example.

In his Master thesis, El-Korashy proves a whole-system form of capability monotonicity: if in a state `s` no capability with permission `p` has authority to address `a`, then the same holds for all future states of `s` [41, Theorem 3.1]. This is a strong assumption that is unlikely to be met in practice. El-Korashy also proves a compartmentalisation result.

He considers a static compartmentalisation setup, where compartments are assigned disjoint private memory, and three memory regions belonging to other compartments that it may respectively jump to, load data from, or store data to. Compartments cannot be created, adjusted, or destroyed, and compartments cannot exchange capabilities: they can only load and store capabilities to their own private memory, and domain transitions clear all capability registers. El-Korashy defines when a state respects a compartmentalisation setup with 16 clauses that, for example, require that the capabilities with the PermitLoadCapability permission in the private memory of a compartment only have authority to that private memory. He then proves that if a state respects a compartmentalisation setup, all future states do too [41, Theorem 5.5]. This results crosses domain boundaries, while our compartmentalisation theorems (Theorem 4.4–4.7) do not. On the other hand, our theorems are not restricted to a static compartmentalisation setup, and also describe which memory locations remain unchanged during the execution of a compartment. Furthermore, El-Korashy uses pen-and-paper proofs, while our results are mechanised. Finally, they base their proofs on an idealised capability machine, inspired by CHERI, but much simpler.

Roe [159, §9.4] describes how CHERI's capability system can be used to protect a reference monitor from untrusted code. With a combination of prose and temporal logic, Roe defines what correctness means in this example and under which assumptions it should hold. More specifically, he defines `SecureState` and `TCBEntryState` that respectively capture when a state is safe to be used by untrusted code and when a state is an entry state of the reference monitor, and conjectures that `SecureState` implies `TCBEntryState R SecureState`, where `R` is the *release* operator. With eight prose lemmas, Roe highlights aspects of CHERI that need to be considered, such as unpredictable behaviour and the TLB state, but he does not prove that CHERI-MIPS satisfies these. Unfortunately, his conjecture is flawed. It requires that `SecureState` still holds at entry states, which is false in CHERI-MIPS, but which is also undesirable because it implies that the reference monitor lost control over its own memory. Furthermore, the conjecture does not capture that untrusted code cannot change the memory of the reference monitor, which was the original intention. Nevertheless, Roe's conjecture is informative and inspired the statement of our simple compartmentalisation scenario (see Section 4.5).

Klein et al. [77, 78] formally proved functional correctness of seL4, a general-purpose microkernel, which uses a capability system to authorise access to system resources. This capability system differs significantly from CHERI's: seL4 is a software capability system where capabilities are managed by the kernel and stored in kernel objects, while CHERI's capabilities can be stored alongside data (see partitioned and tagged approach in Section 1.3.1). Furthermore, seL4 explicitly tracks capability derivations by storing the derivation tree for each capability, which makes revocation easier, but puts a limit on the depth of derivations; seL4 authorises memory accesses per page frame, which are typically

4KB in size, while CHERI offers fine-grained memory protection; and seL4 only supports operations that result in a capability whose authority is contained in the authority of its operands, while CHERI supports non-monotonic operations such as sealing, unsealing, and invocation (seL4 uses the term *invocation* for an unrelated concept). Originally, seL4 was verified from an abstract specification down to its C implementation, but Sewell et al. [136] extended the proof down to the binary level for Arm, based on the Armv7 model that Fox et al. developed in L3 [55, 53]. Sewell et al. [135] proved two security properties over seL4's abstract specification, namely *authority confinement* and *integrity*. Authority confinement is broadly similar to our Theorem 4.4 (Monotonicity of available capabilities), as both bound the authority of the compartment that is in control, and integrity is broadly similar to our Theorems 4.5, 4.6, and 4.7, as both show that a compartment cannot modify parts of the state it does not have authority to. Murray et al. [100] proved another security property for seL4, namely intransitive non-interference, and Blackham et al. [15] performed a sound worst-case execution time analysis for seL4, necessary for real-time systems.

EROS [140] is a capability-based microkernel that influenced the design of seL4, and their capability systems are similar. Shapiro et al. [139] proved *confinement* for an idealisation of EROS's capability system. For a set `E` of compartments, they characterise the objects that are mutable by `E` based on a starting state `s`, and they characterise the objects that are mutated by `E` during an execution trace that starts at `s`. Confinement then states that the mutated objects are mutable in `s`. The structure of this result is similar to our definition of available capabilities based on a starting state, and Theorem 4.4 that shows that capabilities obtained during an execution trace are available in the starting state. Doerrie [37, chapter 12] identified minor and major flaws in Shapiro et al.'s pen-and-paper proof of confinement, illustrating the need for proof mechanisation. Doerrie continues to give a mechanised proof of confinement in SDM, an idealised capability system based on Shapiro et al.'s capability system.

There is extensive literature on object capabilities, going back to the 1960s [29]. Here we only discuss some recent work. In an object capability language, an object needs a capability to send a message to another object, and sending messages is the only way it can cause external effects. An object may encapsulate an inner object by forwarding messages to it in a restricted way. To reason about such cases, Devriese et al. [34] use a step-indexed Kripke logical relation, supporting evolvable invariants on shared data structures. Swasey et al. [150] create a program logic for reasoning about encapsulation with object capabilities. One can use this logic to show that a program only returns *low-integrity* values, which do not give access to private state, and Swasey et al. prove that this implies robust safety, which means that the program can be safely run in an untrusted environment. CHERI's capability system has some aspects of the object capability model: using capability invocation to transfer control over an execution to another compartment

can be viewed as sending a message to another object. However, in CHERI there are other ways a compartment can cause external effects. For example, compartments can write to memory that other compartments may read, assuming they are set up with shared memory. While hardware support is not necessary to implement an object capability language, CHERI's sealing mechanism may be used for this purpose. Our definition of available capabilities (see Section 4.2) specifies which object types a compartment has (in the form of sealed capabilities), which object types it can unseal itself, and which object types it can invoke.

There is also extensive literature comparing capabilities to access control, typically citing the confused deputy problem as the differentiating example (see Section 1.3.1). Rajani et al. [118] formally define when a program is free of confused deputy attacks (CDAs), and show that their capability system indeed protects against some CDAs, but, surprisingly, not all CDAs. They add provenance tracking to their capability system to protect against the remaining CDAs. We refer to their detailed overview [118, §6] for past research that compares capabilities to access control.

There is research that investigates capability systems with different fundamental actions. For example, capability systems typically allow a component to grant permissions to another component, but *take-grant* capability systems also allow components to take permissions, possibly through newly created components. There is a long history of reasoning about the flow of permissions in such systems. For example, Budd et al. [18] and Snyder [147] characterise when a component can obtain a permission with the help of conspirators that do not possess that permission themselves.

Then there is work on secure compilation that uses a capability system as a target language. For example, Van Strydonck et al. [155] develop a fully-abstract compiler that dynamically enforces separation logic contracts, and El-Korashy et al. [42] create a fully-abstract compiler that provides security guarantees for partial programs, which may later be linked to potentially malicious code. The latter is based on a capability system inspired by CHERI, but highly idealised.

Leaving formal properties aside, we refer to Levy [84] for a detailed comparison of capability systems up to 1984, and we refer to CHERI's documentation [165, chapter 13] for an overview of the capability systems that influenced its design.

7.3 Follow-on work

Building on our work, Bauereiss et al. [11] proved Reachable Capability Monotonicity for Morello, a prototype CHERI extension of Armv8-A. Their results are based on the Sail model of Morello, which has been automatically translated [6] from Arm's authoritative specification in ASL. Like L3, Sail is a domain specific language for architecture

specifications (see Section 1.4.1), but it captures semantics differently: L3 produces a monolithic state update function, while Sail uses intra-instruction traces of *effects*, exposing the individual memory and (capability) register accesses of instructions. Although designed for a different purpose, effects are broadly similar to our abstract actions, with two notable differences. First, effects are more granular than abstract actions. For example, reading a capability register and writing a restricted version to another register are two distinct effects, while we model this with one action. Second, effects do not specify which kind of capability manipulation took place. For example, writing back the result of loading, restricting, (un)sealing, and invoking a capability is all captured with the same register-write effect, while we model this with different actions.

By defining security properties directly on effect traces, Bauereiss et al. do not need to annotate instructions, which we do for the 56 instructions that manipulate capabilities or access memory, out of CHERI-MIPS's 197 total instructions (see Section 3.6). While our approach costs more effort, it enables stronger security properties: the annotations make the intention of instructions explicit, and our properties require that the specification follows these intentions, which is not possible with Bauereiss et al.'s approach. For example, if an instruction that is supposed to store a capability to memory, writes it to a register instead, their properties still hold, while our properties would catch this.

Following our general technique, Bauereiss et al. factored their security properties through an abstraction, with a proof that Morello satisfies the abstraction, and higher level security properties proven above the abstraction. Their proof that Morello satisfies their abstraction is more involved than our proof that CHERI-MIPS satisfies our abstraction, as Morello is more complicated: its specification is ten times larger and it contains aspects that CHERI-MIPS does not, such as hardware managed address translation. Even though our abstraction is not intrinsically tied to CHERI-MIPS, it refers to MIPS-specific aspects such as branch delay slots, and therefore does not apply to other CHERI ISAs. Bauereiss et al.'s abstraction, on the other hand, is phrased in a more architecture-independent way, and preliminary work suggests that it also applies to CHERI-RISC-V.

Bauereiss et al. use their abstraction to prove Reachable Capability Monotonicity, which is conceptually the same as our Theorem 4.4 (Monotonicity of available capabilities), except it also covers sentry capabilities, which CHERI did not support at the time of our proofs.

Chapter 8

Conclusion

In the introduction we identified two fundamental problems that cause the majority of security vulnerabilities. First, mainstream engineering methods are not suited to find small bugs in corner cases. Second, mainstream hardware architectures and C/C++ language abstractions provide only coarse-grained memory protection, which lets these small bugs escalate to serious security vulnerabilities. Many solutions have been proposed for the second problem, but despite these the problem remains: general defences have compatibility and performance issues that prevent adoption in practice, while defences against specific attacks led to an ongoing arms race (see Section 1.2).

CHERI, the context of our thesis, aims to address the second problem without the prohibitive downsides of the other proposed solutions. It achieves this by extending commodity architectures with a *capability system*, providing fine-grained memory protection and scalable software compartmentalisation, with low overhead and a gradual adoption path (see Section 1.3). CHERI has been initially developed as CHERI-MIPS, with later work on other architectures.

Before our work, the first fundamental problem applied to CHERI-MIPS: its engineering methods could not give assurance that CHERI-MIPS satisfied its intended security properties, and indeed, CHERI-MIPS contained some serious vulnerabilities (see Chapter 6). Moreover, the intended security properties were not even precisely stated (see Section 1.4). We addressed this problem in our thesis: we showed that formal statement and mechanised proof of security properties can be made feasible for a production-scale capability-enhanced architecture, namely CHERI-MIPS (see Section 1.5). The formal statement of security properties makes it clearer what "correctness and security" means for CHERI-MIPS. Furthermore, the mechanised proofs give a level of confidence in the correctness and security of CHERI-MIPS that cannot be achieved with mainstream engineering methods.

We factored our security properties through a new abstraction layer of CHERI-MIPS. This abstraction explains execution steps in terms of nine abstract actions, one for each

kind of memory access, capability manipulation, and security domain transition (see Chapter 3). Before our work, it was impractical to reason about compromised or malicious CHERI-MIPS executions, as one would have had to consider all the 200-odd CHERI-MIPS instructions and their possible behaviour. Reasoning above our abstraction, on the other hand, is much simpler. Because we proved that our abstraction can simulate CHERI-MIPS (see Theorem 3.17 on page 111), any safety property proven above our abstraction also holds for CHERI-MIPS.

We followed this approach ourselves: we characterised which capabilities can be accessed or constructed by potentially compromised or malicious code and which memory locations it can overwrite until it transitions to another security domain. We used this to prove the correctness of a simple compartmentalisation scenario, in which CHERI's capability system is used to isolate a component from the rest of the system (see Chapter 4). Both these results were proven above our abstraction, and carried over to CHERI-MIPS because of our simulation theorem.

Proving our properties for CHERI-MIPS was challenging because of the size of the architecture, its easy-to-miss corner cases, and the fact that the architecture keeps evolving (see Chapter 5). Our results are based on a full, non-idealised, sequential specification of CHERI-MIPS, which is complete enough to boot an operating system (see Chapter 2). All our proofs are mechanised in Isabelle/HOL.

By showing that all this is possible, we hope that our work encourages the development of formal security properties for other production-scale architectures. Indeed, building on our work, the formal verification of security properties is an important part of the Morello program, which is developing the eponymous prototype CHERI extension of the Armv8-A architecture, along with a processor implementation, development board, and software. The security properties that have been developed so far for Morello follow our general approach: they are factored through an abstraction, with a proof that Morello satisfies the abstraction, and higher level security properties proven above the abstraction (see Section 7.3).

Appendix A

Additional formal definitions

A.1 The Isabelle/HOL machine state

Below we include an alphabetical list of the projection functions that Isabelle generates for the type `State` and its subtypes. We only include the projection functions that are necessary to define our security properties. Note that we renamed some fields for readability, as explained in Section 2.4.5.

`_BadInstr r`
: Returns the opcode of the instruction that caused the last exception, as specified in the system register `r`. We define the following abbreviation, with `s` a machine state: `BadInstr s ≡ _BadInstr (CP0 (ProcessorState s))`.

`_BigEndian r`
: Returns `True` if the endianness that is specified in the configuration register `r` is big-endian, as opposed to little-endian. We define the following abbreviation: `BigEndian s ≡ _BigEndian (Config (CP0 (ProcessorState s)))`.
Note that CHERI-MIPS only supports big-endian, so a state `s` is only valid for CHERI-MIPS if `BigEndian s = True`.

`_BranchDelay p`
: Returns the branch delay PC of the processor state `p`. We define the following abbreviation: `BranchDelay s ≡ _BranchDelay (ProcessorState s)`. The function either returns `None`, or `Some vAddr` with `vAddr` a virtual address. The latter indicates that the current instruction is in a branch delay slot, and `vAddr` then denotes the address that is jumped to.

`BranchDelayPCC s`
: Returns the branch delay PCC of the machine state `s`. It returns either `None`, or `Some (vAddr, cap)` with `vAddr` a virtual address and `cap` a capability. The latter

indicates that the current instruction is in a branch delay slot. In that case, `vAddr` and `cap` denote respectively the address and the capability that is jumped to.

`_BranchTo p`

This is ghost state that is used internally in the L3 specification. Like `_BranchDelay`, it returns either `None`, or `Some vAddr`. It is used in the specification of branch instructions: these first update `BranchTo`, and then at the end of the instruction step they copy `BranchTo` to `BranchDelay` and clear `BranchTo`. We define the following abbreviation: `BranchTo s ≡ _BranchTo (ProcessorState s)`.

`BranchToPCC s`

This is ghost state that relates to `BranchDelayPCC` in the same way as `BranchTo` relates to `BranchDelay`.

`_EPC r`

Returns the exception program counter (EPC) of the system register `r`. We define the following abbreviation: `EPC s ≡ _EPC (CP0 (ProcessorState s))`.

`_ExceptionCode r`

Returns the exception code of the cause register `r`, which corresponds to the type of the last exception that occurred, for example, `0x00` for interrupts, `0x08` for system calls, `0x0C` for arithmetic overflow, et cetera. We define the following abbreviation: `ExceptionCode s ≡ _ExceptionCode (Cause (CP0 (ProcessorState s)))`.

`_ExceptionLevel r`

Returns the exception level of the status register `r`, which is `True` if a hardware exception is being handled. We define the following abbreviation: `ExceptionLevel s ≡ _ExceptionLevel (Status (CP0 (ProcessorState s)))`.

`_ExceptionSignalled p`

Returns whether the processor state `p` indicates that a hardware exception has been raised in the current execution step. We define the following abbreviation: `ExceptionSignalled s ≡ _ExceptionSignalled (ProcessorState s)`.

This field is not architectural state, but is only used internally in the specification. If a previous instruction step raised an exception that has not been handled yet, `ExceptionSignalled s` can be `False` while `ExceptionLevel s` is `True`.

`_GPCapReg s i`

With `i` a 5-bit register index, this returns the `i`-th general purpose capability register. We define the following auxiliary function that always returns the null capability for register 0: `GPCapReg s i ≡ if i = 0 then NullCap else _GPCapReg s i`. Register 26 is called the invoked data capability, for which we define the following abbreviation: `InvokedDataCap s ≡ GPCapReg s 26`.

`_GPR s i`

With `i` a 5-bit register index, this returns the `i`-th general purpose register. We

define the following auxiliary function that always returns 0 for register 0:

```
GPR s i ≡ if i = 0 then 0 else _GPR s i.
```

`L3Exception s`

Returns either `NoException`, or `Unpredictable m` with `m` a string. L3 exceptions should not be confused with architectural exceptions; they are only used to indicate unpredictable behaviour (see Section 2.3.4). We define the following auxiliary function that returns whether a state `s` is unpredictable:

```
IsUnpredictable s ≡
  case L3Exception s
  of NoException ⇒ False
     Unpredictable m ⇒ True
```

`_LoadLinkFlag p`

Returns the load link flag of the processor state `p`. We define the following abbreviation: `LoadLinkFlag s ≡ _LoadLinkFlag (ProcessorState s)`. The function either returns `None`, or `Some b` with `b` a boolean. The value `Some b` indicates that a linked load is in progress, where `b` indicates whether the linked load can be successfully completed by a conditional store. In other words, if `b = False`, then a non-conditional store has written to the address of the linked load.

`Mem s a`

Here, `a` is a 35-bit word that is interpreted as the upper bits of a physical address, which corresponds to a capability-sized and -aligned region of memory. `Mem s a` returns the memory contents of that region, which can either be `RawMemValue w` with `w` a 256-bit word, or `CapMemValue cap` with `cap` a capability.

`_PC p`

Returns the program counter (PC) of the processor state `p`. We define the following abbreviation: `PC s ≡ _PC (ProcessorState s)`.

`PCC s`

Returns the program counter capability (PCC).

`ProcessorState s`

Returns the state of the currently running processor core. The specification does not cover multi-core machines, so the distinction between the overall machine state and the processor state is moot.

`_ReverseEndian r`

Returns whether the status register `r` specifies that the endianness should be reversed for user mode instructions. We define the following abbreviation:

`ReverseEndian s ≡ _ReverseEndian (Status (CP0 (ProcessorState s)))`. CHERI-MIPS does not support reversing the endianness, so a state `s` is only valid for CHERI-MIPS if `ReverseEndian s = False`.

`SpecialCapReg s i`

With `i` a 5-bit register index, this returns the capability in the `i`-th special capability register. There are three special registers that are used in our security properties. We define the following abbreviations for them:

`DefaultDataCap s` ≡ `SpecialCapReg s 0`,

`KernelCodeCap s` ≡ `SpecialCapReg s 29`, and

`EPCC s` ≡ `SpecialCapReg s 31`, where EPCC stands for the exception PCC.

A.2 Example export of an auxiliary function

To give an impression of the definitions that our proof tactics need to handle, we include the Isabelle export of `SignalException` here (see Figure 2.7 on page 59 for the L3 source). The exported definition does not need to be understood in detail to follow the contribution of this thesis.

```
SignalException t ≡
 do {
   v ← ReadCP0;
   do {
     v0 ← NextUnknown "BadInstrP";
     Bind (do {
             v ← do {
                 v0 ← Return (Undefined v0);
                 Return (v, v0)
               };
             Return ((First v)(BadInstrP := Second v))
           })
      WriteCP0
   };
   v ← ReadCP0;
   _4 ← do {
       b ← do {
           v ← do {
               v ← Return (Status v);
               Return (_ExceptionLevel v)
             };
           Return (not v)
         };
       if b
       then do {
              v ← ReadBranchDelay;
              b ← do {
                  b ← Return (IsSome v);
                  if b
                  then Return True
                  else do {
                         v ← ReadState BranchDelayPCC;
                         Return (IsSome v)
                       }
                }
            };
            if b
            then do {
```

```
                    v ← ReadCP0;
                    _1 ← do {
                        v0 ← ReadPC;
                        Bind (do {
                                v ← do {
                                    v0 ← Return (v0 - 4);
                                    Return (v, v0)
                                  };
                                Return ((First v)(_EPC := Second v))
                              })
                         WriteCP0
                      };
                    v ← ReadCP0;
                    _2 ← WriteCP0 (v(Cause := (Cause v)(BD := True)));
                    v ← ReadState LastInstruction;
                    b ← Return (IsSome v);
                    if b
                    then do {
                           x ← ReadCP0;
                           v ← ReadState LastInstruction;
                           Bind (do {
                                   v ← do {
                                       v ← Return (The v);
                                       Return (x, v)
                                     };
                                   Return ((First v)(BadInstrP := Second v))
                                 })
                            WriteCP0
                         }
                    else Return ()
                  }
             else do {
                    v ← ReadCP0;
                    _3 ← do {
                        v0 ← ReadPC;
                        Bind (do {
                                v ← Return (v, v0);
                                Return ((First v)(_EPC := Second v))
                              })
                         WriteCP0
                      };
                    x ← ReadCP0;
                    WriteCP0 (x(Cause := (Cause x)(BD := False)))
                  }
          }
      else Return ()
    };
  v ← if t = XTLBRefillL or t = XTLBRefillS
      then do {
             v ← ReadCP0;
             v ← do {
                 v ← Return (Status v);
                 Return (_ExceptionLevel v)
               };
             Return (not v)
           }
      else Return False;
  vectorOffset ← if v
                 then Return 128
```

```
                  else do {
                         b ← if t = C2E
                              then do {
                                     v ← ReadState CapCause;
                                     b ← do {
                                         v ← Return (CapExceptionCode v);
                                         Return (v = 5)
                                       };
                                     if b
                                     then Return True
                                     else do {
                                            v ← ReadState CapCause;
                                            v ← Return (CapExceptionCode v);
                                            Return (v = 6)
                                          }
                                   }
                              else Return False;
                         Return if b
                                then 640
                                else 384
                       };
    v ← ReadState CurrentInstruction;
    _5 ← do {
        b ← Return (IsSome v);
        if b
        then do {
               x ← ReadCP0;
               v ← ReadState CurrentInstruction;
               Bind (do {
                       v ← do {
                           v ← Return (The v);
                           Return (x, v)
                         };
                       Return ((First v)(_BadInstr := Second v))
                     })
                WriteCP0
             }
        else Return ()
      };
    v ← ReadCP0;
    _6 ← WriteCP0 (v(Cause := (Cause v)(_ExceptionCode := ReadExceptionCode t)));
    v ← ReadCP0;
    vectorBase ← do {
        b ← do {
            v ← Return (Status v);
            Return (BEV v)
          };
        Return if b
               then 18446744072631616000
               else 18446744071562067968
      };
    _7 ← WriteBranchDelay None;
    _8 ← WriteBranchTo None;
    _9 ← UpdateState (λs. s(BranchDelayPCC := None));
    _10 ← UpdateState (λs. s(BranchToPCC := None));
    _11 ← WriteExceptionSignalled True;
    v ← ReadPCC;
    ExtendState v (do {
                     v ← TrimState ReadPCC;
```

```
                _12 ← do {
                    b ← do {
                        v ← do {
                            v ← do {
                                v0 ← TrimState ReadPC;
                                Return (v, v0)
                              };
                            Return (CanRepOffset v)
                          };
                        Return (not v)
                      };
                    if b
                    then do {
                           v ← TrimState ReadPCC;
                           v ← do {
                               v ← do {
                                   v ← do {
                                       v ← Return (Base v);
                                       v0 ← TrimState ReadPC;
                                       Return (v + v0)
                                     };
                                   Return (NullCap, v)
                                 };
                               Return (SetOffset v)
                             };
                           UpdateState (λs. (v, Second s))
                         }
                    else do {
                           v ← ReadState First;
                           v ← do {
                               v ← do {
                                   v0 ← TrimState ReadPC;
                                   Return (v, v0)
                                 };
                               Return (SetOffset v)
                             };
                           UpdateState (λs. (v, Second s))
                         }
                  };
                v ← TrimState ReadCP0;
                _13 ← do {
                    b ← do {
                        v ← do {
                            v ← Return (Status v);
                            Return (_ExceptionLevel v)
                          };
                        Return (not v)
                      };
                    if b
                    then do {
                           v ← ReadState First;
                           TrimState (WriteEPCC v)
                         }
                    else Return ()
                  };
                v ← TrimState ReadKCC;
                _14 ← TrimState (WritePCC v);
                v ← TrimState ReadPCC;
                _15 ← do {
```

```
                    v ← do {
                        v ← Return (Base v);
                        Return (WordCat (ExtractWord 63 30 vectorBase)
                                 (ExtractWord 29 0 vectorBase + vectorOffset) -
                                v)
                      };
                    TrimState (WritePC v)
                  };
                v ← TrimState ReadCP0;
                _16 ← TrimState (WriteCP0 (v(Status := (Status v)(_ExceptionLevel := True))));
                Return ()
              })
}
```

A.3 Security properties

Here we include all the definitions from our proof development that are necessary to state our security properties, but that are omitted or explained in prose in the main text. The entire proof development is available online [108].

Definition A.1. The function `ValuePart` projects a monadic function `m` to a function from states to values. Formally: `ValuePart m s ≡ First (m s)`

Definition A.2. The function `StatePart` projects a monadic function `m` to a function from starting states to resulting states. Formally: `StatePart m s ≡ Second (m s)`

Definition A.3. The function `MemTag` returns whether the capability-sized and -aligned region of memory at address `a` has a tag. Formally: `MemTag s a ≡ Tag (MemCap s a)`

Definition A.4. The function `ActionAuthority` takes an abstract action `action` and returns `Some r` if the action uses the register `r` as authority, and returns `None` otherwise. Formally:

```
ActionAuthority action ≡
  case action
  of LoadDataAction auth a l ⇒ Some auth
     StoreDataAction auth a l ⇒ Some auth
     RestrictCapAction r r' ⇒ None
     LoadCapAction auth a r ⇒ Some auth
     StoreCapAction auth r a ⇒ Some auth
     SealCapAction auth r r' ⇒ Some auth
     UnsealCapAction auth r r' ⇒ Some auth
```

Definition A.5. The function `ActionSources` takes an abstract action `action` and returns the set of capability locations that the action uses as the source of a capability manipulation. This set is empty for the action about loading data. For the other actions,

this set consists of the source register or the source address. Formally:

```
ActionSources action ≡
  case action
  of LoadDataAction auth a ln ⇒ {}
     StoreDataAction auth a ln ⇒ {LocMem (GetCapAddress a)}
     RestrictCapAction r r' ⇒ {LocReg r}
     LoadCapAction auth a r ⇒ {LocMem a}
     StoreCapAction auth r a ⇒ {LocReg (RegGeneral r)}
     SealCapAction auth r r' ⇒ {LocReg (RegGeneral r)}
     UnsealCapAction auth r r' ⇒ {LocReg (RegGeneral r)}
```

Definition A.6. The function `SpecialRegisterParameters` takes an abstract action `action` and returns the set of special capability registers that it uses. An action can use registers either as authority, as source, or as target. Formally:

```
SpecialRegisterParameters action ≡
  case ActionAuthority action
  of None ⇒ {}
     Some (RegSpecial i) ⇒ {i}
     Some _ ⇒ {} ∪
  {i |i. LocReg (RegSpecial i) ∈ ActionSources action} ∪
  {i |i. LocReg (RegSpecial i) ∈ ActionTargets action}
```

Definition A.7. The function `ExceptionPCs` returns the set of entry addresses of exception handlers in MIPS. These addresses are calculated by combining a base with an offset, as described formally below:

```
ExceptionPCs ≡
  let vectorBases = {18446744072631616000, 18446744071562067968} in
  let vectorOffsets = {128, 384, 640} in
  {WordCat (Slice 30 vectorBase) (UnsignedCast vectorBase + vectorOffset)
|vectorBase vectorOffset.
   vectorBase ∈ vectorBases ∧ vectorOffset ∈ vectorOffsets}
```

Definition A.8. The function `TranslateAddresses` takes a set of virtual addresses and, for each address that can be translated, returns the translation. Formally:

```
pAddr ∈ TranslateAddresses vAddrs t s ≡
  exists vAddr. vAddr ∈ vAddrs
                and TranslateAddr (vAddr, t) s = Some pAddr
```

Definition A.9. The function `TranslateCapAddresses` takes a set of virtual addresses and returns the 35-bit addresses of the capability-sized and -aligned regions of memory that contain the translations of these virtual addresses. Formally:

```
a ∈ TranslateCapAddresses vAddrs t s ≡
  exists pAddr. pAddr ∈ TranslateAddresses vAddrs t s
                and a = GetCapAddress pAddr
```

Definition A.10. The function `RegisterIsAlwaysAccessible` returns whether a register `r` can be accessed without any authority. This is true for the PCC, the branch delay PCC, and the general purpose capability registers. It is also true for the DDC and the TLSC, which are respectively the special capability registers 0 and 1. Formally:

```
RegisterIsAlwaysAccessible r ≡
  case r
  of RegSpecial r ⇒ r = 0 or r = 1
     _ ⇒ True
```

Mapping instructions that restrict capabilities

In Section 3.6.1 we describe how we map instructions that restrict capabilities to our abstraction. In that section, we define this mapping formally for three instructions, namely CAndPerm, CMOVZ, and CJR. Here we define the mapping of the other instructions that restrict capabilities.

Definition A.11. We map the CSetOffset instruction as follows:

```
CSetOffsetActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.12. We map the CIncOffset instruction as follows:

```
CIncOffsetActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.13. We map the CIncOffsetImmediate instruction as follows:

```
CIncOffsetImmediateActions (cd, cb, increment) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.14. We map the CFromPtr instruction as follows:

```
CFromPtrActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.15. We map the CBuildCap instruction as follows:

```
CBuildCapActions (cd, cb, ct) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.16. We map the CClearTag instruction as follows:

```
CClearTagActions (cd, cb) s ≡
  {RestrictCapAction (RegGeneral cd) (RegGeneral cd)}
```

Definition A.17. We map the CClearHi instruction as follows:

```
CClearHiActions m s ≡
  {RestrictCapAction (RegGeneral cd) (RegGeneral cd) |cd.
   Bit m (WordToNat (cd - 16))}
```

Definition A.18. We map the CClearLo instruction as follows:

```
CClearLoActions m s ≡
  {RestrictCapAction (RegGeneral cd) (RegGeneral cd) |cd.
   Bit m (WordToNat cd)}
```

Definition A.19. We map the CCopyType instruction as follows:

```
CCopyTypeActions (cd, cb, ct) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.20. We map the CGetPCC instruction as follows:

```
CGetPCCActions cd s ≡
  {RestrictCapAction RegPCC (RegGeneral cd)}
```

Definition A.21. We map the CGetPCCSetOffset instruction as follows:

```
CGetPCCSetOffsetActions (cd, rs) s ≡
  {RestrictCapAction RegPCC (RegGeneral cd)}
```

Definition A.22. We map the CMove instruction as follows:

```
CMoveActions (cd, cs) s ≡
  {RestrictCapAction (RegGeneral cs) (RegGeneral cd)}
```

Definition A.23. We map the CMOVN instruction as follows:

```
CMOVNActions (cd, cb, rt) s ≡
  if GPR s rt ≠ 0
  then {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
  else {}
```

Definition A.24. We map the CReadHwr instruction as follows:

```
CReadHwrActions (cd, selector) s ≡
  {RestrictCapAction (RegSpecial selector) (RegGeneral cd)}
```

Definition A.25. We map the CWriteHwr instruction as follows:

```
CWriteHwrActions (cb, selector) s ≡
  {RestrictCapAction (RegGeneral cb) (RegSpecial selector)}
```

Definition A.26. We map the CJALR instruction as follows:

```
1 CJALRActions (cd, cb) s ≡
2   if cb = cd
3   then {RestrictCapAction RegPCC (RegGeneral cd),
4         RestrictCapAction RegPCC RegBranchDelayPCC}
5   else {RestrictCapAction RegPCC (RegGeneral cd),
6         RestrictCapAction (RegGeneral cb) RegBranchDelayPCC}
```

Definition A.27. We map the CSetBounds instruction as follows:

```
CSetBoundsActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.28. We map the CSetBoundsExact instruction as follows:

```
CSetBoundsExactActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Definition A.29. We map the CSetBoundsImmediate instruction as follows:

```
CSetBoundsImmediateActions (cd, cb, rt) s ≡
  {RestrictCapAction (RegGeneral cb) (RegGeneral cd)}
```

Mapping instructions that load data

In Section 3.6.3 we describe how we map instructions that load data to our abstraction. Here, we define their mapping formally.

Definition A.30. The function `CLoadVirtualAddress` returns the virtual address that the CLoad instruction loads from. Formally:

```
CLoadVirtualAddress cb rt offset t s ≡
  Address (GPCapReg s cb) + GPR s rt +
  (SignedCast offset << WordToNat t)
```

Definition A.31. The function `CLoadPhysicalAddress` returns the physical address that the CLoad instruction loads from. Formally:

```
CLoadPhysicalAddress cb rt offset t s ≡
  TranslateAddr (CLoadVirtualAddress cb rt offset t s, Load) s
```

Definition A.32. We map the CLoad instruction as follows:

```
1 CLoadActions (rd, cb, rt, offset, x, t) s ≡
2   case CLoadPhysicalAddress cb rt offset t s
3   of None ⇒ {}
4      Some a ⇒ let length = 2^WordToNat t in
5                {LoadDataAction (RegGeneral cb) a length}
```

Definition A.33. The function `CLLxVirtualAddress` returns the virtual address that the CLLx instructions load from. Formally:

```
CLLxVirtualAddress cb s ≡ Address (GPCapReg s cb)
```

Definition A.34. The function `CLLxPhysicalAddress` returns the physical address that the CLLx instructions load from. Formally:

```
CLLxPhysicalAddress cb s ≡
  TranslateAddr (CLLxVirtualAddress cb s, Load) s
```

Definition A.35. We map the CLLx instructions as follows:

```
CLLxActions (rd, cb, stt) s ≡
  case CLLxPhysicalAddress cb s
  of None ⇒ {}
     Some a ⇒
       let length = 2^WordToNat (stt AND Mask 2) in
       {LoadDataAction (RegGeneral cb) a length}
```

Definition A.36. The function `LegacyLoadVirtualAddress` returns the virtual address that legacy load instructions load from. Formally:

```
LegacyLoadVirtualAddress b offset s ≡
  Address (DefaultDataCap s) + GPR s b + SignedCast offset
```

Definition A.37. The function `LegacyLoadPhysicalAddress` returns the physical address that legacy load instructions load from. Formally:

```
LegacyLoadPhysicalAddress b offset s ≡
  TranslateAddr (LegacyLoadVirtualAddress b offset s, Load) s
```

Definition A.38. We map legacy load instructions as follows:

```
LegacyLoadActions b offset l s ≡
  case LegacyLoadPhysicalAddress b offset s
  of None ⇒ {}
     Some a ⇒ {LoadDataAction (RegSpecial 0) a l}
```

Definition A.39. We map the LB instruction as follows:

```
LBActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 1 s
```

Definition A.40. We map the LBU instruction as follows:

```
LBUActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 1 s
```

Definition A.41. We map the LH instruction as follows:

```
LHActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 2 s
```

Definition A.42. We map the LHU instruction as follows:

```
LHUActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 2 s
```

Definition A.43. We map the LW instruction as follows:

```
LWActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 4 s
```

Definition A.44. We map the LWU instruction as follows:

```
LWUActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 4 s
```

Definition A.45. We map the LD instruction as follows:

```
LDActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 8 s
```

Definition A.46. We map the LL instruction as follows:

```
LLActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 4 s
```

Definition A.47. We map the LLD instruction as follows:

```
LLDActions (b, rt, offset) s ≡
  LegacyLoadActions b offset 8 s
```

Definition A.48. We map the LWL instruction as follows:

```
LWLActions (b, rt, offset) s ≡
  let vAddr = LegacyLoadVirtualAddress b offset s in
  let length = (NOT UnsignedCast vAddr AND Mask 2) + 1 in
  case TranslateAddr (vAddr, Load) s
  of None ⇒ {}
     Some pAddr ⇒ {LoadDataAction (RegSpecial 0) pAddr length}
```

Definition A.49. We map the LWR instruction as follows:

```
LWRActions (b, rt, offset) s ≡
  let vAddr = LegacyLoadVirtualAddress b offset s in
  let length = (UnsignedCast vAddr AND Mask 2) + 1 in
  case TranslateAddr (vAddr AND NOT Mask 2, Load) s
  of None ⇒ {}
     Some pAddr ⇒ {LoadDataAction (RegSpecial 0) pAddr length}
```

Definition A.50. We map the LDL instruction as follows:

```
LDLActions (b, rt, offset) s ≡
  let vAddr = LegacyLoadVirtualAddress b offset s in
  let length = (NOT UnsignedCast vAddr AND Mask 3) + 1 in
  case TranslateAddr (vAddr, Load) s
  of None ⇒ {}
     Some pAddr ⇒ {LoadDataAction (RegSpecial 0) pAddr length}
```

Definition A.51. We map the LDR instruction as follows:

```
LDRActions (b, rt, offset) s ≡
  let vAddr = LegacyLoadVirtualAddress b offset s in
  let length = (UnsignedCast vAddr AND Mask 3) + 1 in
  case TranslateAddr (vAddr AND NOT Mask 3, Load) s
  of None ⇒ {}
     Some pAddr ⇒ {LoadDataAction (RegSpecial 0) pAddr length}
```

Mapping instructions that store data

In Section 3.6.3 we describe how we map instructions that store data to our abstraction. In that section, we define this mapping formally for four instructions, namely CStore, SD, SDL, and SDR. Here we define the mapping of the other instructions that store data.

Definition A.52. The function `CSCxVirtualAddress` returns the virtual address that the CSCx instructions store to. Formally:

```
CSCxVirtualAddress cb s ≡ Address (GPCapReg s cb)
```

Definition A.53. The function `CSCxPhysicalAddress` returns the physical address that the CSCx instructions store to. Formally:

```
CSCxPhysicalAddress cb s ≡
  TranslateAddr (CSCxVirtualAddress cb s, Store) s
```

Definition A.54. We map the CSCx instructions as follows:

```
CSCxActions (rs, cb, rd, t) s ≡
  case CSCxPhysicalAddress cb s
  of None ⇒ {}
     Some a ⇒ let length = 2^WordToNat t in
               {StoreDataAction (RegGeneral cb) a length}
```

Definition A.55. The function `LegacyStoreVirtualAddress` returns the virtual address that the legacy store instructions store to. Formally:

```
LegacyStoreVirtualAddress b offset s ≡
  Address (DefaultDataCap s) + GPR s b + SignedCast offset
```

Definition A.56. The function `LegacyStorePhysicalAddress` returns the physical address that the legacy store instructions store to. Formally:

```
LegacyStorePhysicalAddress b offset s ≡
  TranslateAddr (LegacyStoreVirtualAddress b offset s, Store) s
```

The function `LegacyStoreActions` is defined in the main text (see Section 3.6.3).

Definition A.57. We map the SB instruction as follows:

```
SBActions (b, rt, offset) s ≡
  LegacyStoreActions b offset 1 s
```

Definition A.58. We map the SH instruction as follows:

```
SHActions (b, rt, offset) s ≡
  LegacyStoreActions b offset 2 s
```

Definition A.59. We map the SW instruction as follows:

```
SWActions (b, rt, offset) s ≡
  LegacyStoreActions b offset 4 s
```

Definition A.60. We map the SC instruction as follows:

```
SCActions (b, rt, offset) s ≡
  LegacyStoreActions b offset 4 s
```

Definition A.61. We map the SCD instruction as follows:

```
SCDActions (b, rt, offset) s ≡
  LegacyStoreActions b offset 8 s
```

Definition A.62. We map the SWL instruction as follows:

```
SWLActions (b, rt, offset) s ≡
  let vAddr = LegacyStoreVirtualAddress b offset s in
  let length = (NOT UnsignedCast vAddr AND Mask 2) + 1 in
  case TranslateAddr (vAddr, Store) s
  of None ⇒ {}
     Some pAddr ⇒ {StoreDataAction (RegSpecial 0) pAddr length}
```

Definition A.63. We map the SWR instruction as follows:

```
SWRActions (b, rt, offset) s ≡
  let vAddr = LegacyStoreVirtualAddress b offset s in
  let length = (UnsignedCast vAddr AND Mask 2) + 1 in
  case TranslateAddr (vAddr AND NOT Mask 2, Store) s
  of None ⇒ {}
     Some pAddr ⇒ {StoreDataAction (RegSpecial 0) pAddr length}
```

Mapping instructions that load capabilities

In Section 3.6.4 we describe how we map instructions that load capabilities to our abstraction. In that section, we define the mapping formally for the CLC instruction. Here we define the auxiliary functions that this mapping depends on, and we define the mapping of the other instruction that loads capabilities, namely CLLC.

Definition A.64. The function `CLCVirtualAddress` returns the virtual address that the CLC instruction loads from. Formally:

```
CLCVirtualAddress cb rt offset s ≡
  Address (GPCapReg s cb) + GPR s rt +
  16 * SignedCast offset
```

Definition A.65. The function `CLCPhysicalAddress` returns the physical address that the CLC instruction loads from. Formally:

```
CLCPhysicalAddress cb rt offset s ≡
  TranslateAddr (CLCVirtualAddress cb rt offset s, Load) s
```

The function `CLCActions` is defined in the main text (see Section 3.6.4).

Definition A.66. The function `CLLCVirtualAddress` returns the virtual address that the CLLC instruction loads from. Formally:

```
CLLCVirtualAddress cb s ≡ Address (GPCapReg s cb)
```

Definition A.67. The function `CLLCPhysicalAddress` returns the physical address that the CLLC instruction loads from. Formally:

```
CLLCPhysicalAddress cb s ≡
  TranslateAddr (CLLCVirtualAddress cb s, Load) s
```

Definition A.68. We map the CLLC instruction as follows:

```
1 CLLCActions (cd, cb) s ≡
2   case CLLCPhysicalAddress cb s
3   of None ⇒ {}
4      Some a ⇒
5        if PermitLoadCapability (GPCapReg s cb)
6        then {LoadCapAction (RegGeneral cb) (GetCapAddress a) cd}
7        else {RestrictCapAction (RegGeneral cd) (RegGeneral cd),
8              LoadDataAction (RegGeneral cb) a 32}
```

Mapping instructions that store capabilities

In Section 3.6.4 we describe how we map instructions that store capabilities to our abstraction. In that section, we define the mapping formally for the CSC instruction. Here we define the auxiliary functions that this mapping depends on, and we define the mapping of the other instruction that stores capabilities, namely CSCC.

Definition A.69. The function `CSCVirtualAddress` returns the virtual address that the CSC instruction stores to. Formally:

```
CSCVirtualAddress cb rt offset s ≡
  Address (GPCapReg s cb) + GPR s rt +
  16 * SignedCast offset
```

Definition A.70. The function `CSCPhysicalAddress` returns the physical address that the CSC instruction stores to. Formally:

```
CSCPhysicalAddress cb rt offset s ≡
  TranslateAddr (CSCVirtualAddress cb rt offset s, Store) s
```

The function `CSCActions` is defined in the main text (see Section 3.6.4).

Definition A.71. The function `CSCCVirtualAddress` returns the virtual address that the CSCC instruction stores to. Formally:

```
CSCCVirtualAddress cb s ≡ Address (GPCapReg s cb)
```

Definition A.72. The function `CSCCPhysicalAddress` returns the physical address that the CSCC instruction stores to. Formally:

```
CSCCPhysicalAddress cb s ≡
  TranslateAddr (CSCCVirtualAddress cb s, Store) s
```

Definition A.73. We map the CSCC instruction as follows:

```
1  CSCCActions (cs, cb, rd) s ≡
2    case LoadLinkFlag s
3    of None ⇒ {}
4       Some True ⇒
5         case CSCCPhysicalAddress cb s
6         of None ⇒ {}
7            Some a ⇒ {StoreCapAction (RegGeneral cb) cs (GetCapAddress a)}
8       Some False ⇒ {}
```

Mapping the entire execution step

In Section 3.6.5 we describe how we map an entire execution step to our abstraction. Here we formally define the auxiliary functions that this mapping depends on.

Definition A.74. The function `InstructionActions` takes a decoded instruction `instr` and returns the actions that the instruction maps to. The function is defined as a large case split over instructions. Formally:

```
1  InstructionActions instr s ≡
2    case instr
3    of COP2 (CHERICOP2 (CBuildCap v)) ⇒ CBuildCapActions v s
4       COP2 (CHERICOP2 (CClearHi v)) ⇒ CClearHiActions v s
5       COP2 (CHERICOP2 (CClearLo v)) ⇒ CClearLoActions v s
6       COP2 (CHERICOP2 (CCopyType v)) ⇒ CCopyTypeActions v s
7       COP2 (CHERICOP2 (CGet (CGetPCC v))) ⇒
8         CGetPCCActions v s
9       COP2 (CHERICOP2 (CGet (CGetPCCSetOffset v))) ⇒
10        CGetPCCSetOffsetActions v s
11      COP2 (CHERICOP2 (CGet _)) ⇒ {}
12      COP2 (CHERICOP2 (CJALR v)) ⇒ CJALRActions v s
13      COP2 (CHERICOP2 (CJR v)) ⇒ CJRActions v s
14      COP2 (CHERICOP2 (CLLC v)) ⇒ CLLCActions v s
```

```
15      COP2 (CHERICOP2 (CLLx v)) ⇒ CLLxActions v s
16      COP2 (CHERICOP2 (CMOVN v)) ⇒ CMOVNActions v s
17      COP2 (CHERICOP2 (CMOVZ v)) ⇒ CMOVZActions v s
18      COP2 (CHERICOP2 (CMove v)) ⇒ CMoveActions v s
19      COP2 (CHERICOP2 (CReadHwr v)) ⇒ CReadHwrActions v s
20      COP2 (CHERICOP2 (CSCC v)) ⇒ CSCCActions v s
21      COP2 (CHERICOP2 (CSCx v)) ⇒ CSCxActions v s
22      COP2 (CHERICOP2 (CSeal v)) ⇒ CSealActions v s
23      COP2 (CHERICOP2 (CSet (CAndPerm v))) ⇒
24        CAndPermActions v s
25      COP2 (CHERICOP2 (CSet (CClearTag v))) ⇒
26        CClearTagActions v s
27      COP2 (CHERICOP2 (CSet (CFromPtr v))) ⇒
28        CFromPtrActions v s
29      COP2 (CHERICOP2 (CSet (CIncOffset v))) ⇒
30        CIncOffsetActions v s
31      COP2 (CHERICOP2 (CSet (CIncOffsetImmediate v))) ⇒
32        CIncOffsetImmediateActions v s
33      COP2 (CHERICOP2 (CSet (CSetBounds v))) ⇒
34        CSetBoundsActions v s
35      COP2 (CHERICOP2 (CSet (CSetBoundsExact v))) ⇒
36        CSetBoundsExactActions v s
37      COP2 (CHERICOP2 (CSet (CSetBoundsImmediate v))) ⇒
38        CSetBoundsImmediateActions v s
39      COP2 (CHERICOP2 (CSet (CSetCause word))) ⇒
40        {}
41      COP2 (CHERICOP2 (CSet (CSetOffset v))) ⇒
42        CSetOffsetActions v s
43      COP2 (CHERICOP2 (CUnseal v)) ⇒ CUnsealActions v s
44      COP2 (CHERICOP2 (CWriteHwr v)) ⇒ CWriteHwrActions v s
45      COP2 (CHERICOP2 _) ⇒ {}
46      ERET ⇒ ERETActions s
47      LDC2 (CHERILDC2 (CLC v)) ⇒ CLCActions v s
48      LWC2 (CHERILWC2 (CLoad v)) ⇒ CLoadActions v s
49      LoadInstr (LB v) ⇒ LBActions v s
50      LoadInstr (LBU v) ⇒ LBUActions v s
51      LoadInstr (LD v) ⇒ LDActions v s
52      LoadInstr (LDL v) ⇒ LDLActions v s
53      LoadInstr (LDR v) ⇒ LDRActions v s
54      LoadInstr (LH v) ⇒ LHActions v s
55      LoadInstr (LHU v) ⇒ LHUActions v s
56      LoadInstr (LL v) ⇒ LLActions v s
57      LoadInstr (LLD v) ⇒ LLDActions v s
58      LoadInstr (LW v) ⇒ LWActions v s
59      LoadInstr (LWL v) ⇒ LWLActions v s
60      LoadInstr (LWR v) ⇒ LWRActions v s
```

```
61      LoadInstr (LWU v) ⇒ LWUActions v s
62      SDC2 (CHERISDC2 (CSC v)) ⇒ CSCActions v s
63      SWC2 (CHERISWC2 (CStore v)) ⇒ CStoreActions v s
64      StoreInstr (SB v) ⇒ SBActions v s
65      StoreInstr (SC v) ⇒ SCActions v s
66      StoreInstr (SCD v) ⇒ SCDActions v s
67      StoreInstr (SD v) ⇒ SDActions v s
68      StoreInstr (SDL v) ⇒ SDLActions v s
69      StoreInstr (SDR v) ⇒ SDRActions v s
70      StoreInstr (SH v) ⇒ SHActions v s
71      StoreInstr (SW v) ⇒ SWActions v s
72      StoreInstr (SWL v) ⇒ SWLActions v s
73      StoreInstr (SWR v) ⇒ SWRActions v s
74      _ ⇒ {}
```

Definition A.75. The function `NextInstruction` returns `None` if fetching the next instruction raises an exception or causes unpredictable behaviour, and otherwise it returns the fetched instruction. Note that `Fetch` returns `Some x` in the latter case, so `NextInstruction` does not need to wrap that result in an option type. Formally:

```
1  NextInstruction s ≡
2    let (instr, s') = Fetch s in
3    if ExceptionSignalled s' or IsUnpredictable s'
4    then None
5    else instr
```

Definition A.76. The function `FetchCCallFastOrOtherInstruction` returns `NoInstruction` if the next instruction cannot be fetched, it returns `CCallFastInstruction cd cd'` if the fetched instruction is CCallFast (cd, cd'), and `OtherInstruction instr` otherwise, where `instr` is the decoded instruction. Formally:

```
FetchCCallFastOrOtherInstruction s ≡
  case NextInstruction s
  of None ⇒ NoInstruction
     Some w ⇒ case Decode w
               of COP2 (CHERICOP2 (CCallFast (cd, cd'))) ⇒
                    CCallFastInstruction cd cd'
                  instr ⇒ OtherInstruction instr
```

Bibliography

[1] William B. Ackerman and William W. Plummer. An implementation of a multiprocessing computer system. In *SOSP 1967: Proceedings of the first ACM symposium on Operating System Principles*, pages 5.1–5.10, 1967. https://doi.org/10.1145/800001.811666.

[2] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Elsevier, 1992. ISBN 1-55558-098-X.

[3] AMD. AMD64 architecture programmer's manual, volumes 1–5. https://www.amd.com/system/files/TechDocs/40332.pdf, 2021. Publication number: 40332.

[4] American National Standards Institute and Computer and Business Equipment Manufacturers Association. *Programming language C*. ANSI, 1990. Document number: X3.159-1989.

[5] ARM Limited. ARMv6-M architecture reference manual. https://developer.arm.com/documentation/ddi0419/e/, 2018. Document number: DDI 0419E (ID070218).

[6] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *POPL 2019: Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019. https://doi.org/10.1145/3290384.

[7] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hriţcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *SP 2015: IEEE Symposium on Security and Privacy*, pages 813–830, 2015. https://doi.org/10.1109/SP.2015.55.

[8] Michael Bailey, Evan Cooke, Farnam Jahanian, David Watson, and Jose Nazario. The Blaster Worm: Then and now. *IEEE Security & Privacy*, 3(4):26–31, 2005. https://doi.org/10.1109/MSP.2005.106.

[9] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005: Revised Lectures of the 4th International Symposium on Formal Methods for Components and Objects*, pages 364–387, 2005. https://doi.org/10.1007/11804192_17.

[10] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The security architecture of the Chromium browser. Technical report, Stanford University, 2008.

[11] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N.M. Watson, and Peter Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture, 2021. Under submission.

[12] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL – lessons learned in formal-logic engineering. In *TPHOLs 1999: International Conference on Theorem Proving in Higher Order Logics*, pages 19–36, 1999. https://doi.org/10.1007/3-540-48256-3_3.

[13] Viktors Berstis. Security and protection of data in the IBM System/38. In *ISCA 1980: Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 245–252, 1980. https://doi.org/10.1145/800053.801932.

[14] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development - Coq'Art: The Calculus of inductive constructions.* Springer, 2004. https://doi.org/10.1007/978-3-662-07964-5.

[15] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *RTSS 2011: Proceedings of the 32nd Real-Time Systems Symposium*, pages 339–348, 2011. https://doi.org/10.1109/RTSS.2011.38.

[16] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS 2011: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011. https://doi.org/10.1145/1966913.1966919.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *CAV 2011: 23rd International Conference on Computer Aided Verification*, pages 463–469, 2011. https://doi.org/10.1007/978-3-642-22110-1_37.

[18] T. Budd and R. Lipton. Inert rights and conspirators in the TAKE/GRANT system. Technical report, Yale Department of Computer Science, 1977. Research report 126.

[19] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack magazine*, 10(56): 5, 2000.

[20] Brian Campbell and Ian Stark. Extracting behaviour from an executable instruction set model. In *FMCAD 2016: Formal Methods in Computer-Aided Design*, pages 33–40, 2016. https://doi.org/10.1109/FMCAD.2016.7886658.

[21] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *ASPLOS 1994: 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327, 1994. https://doi.org/10.1145/195473.195579.

[22] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *CCS 2010: Proceedings of the 17th ACM conference on Computer and Communications Security*, pages 559–572, 2010. https://doi.org/10.1145/1866307.1866370.

[23] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1 (ICFP):24:1–24:30, 2017. https://doi.org/10.1145/3110268.

[24] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://ia.cr/2016/086.

[25] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 98, pages 63–78, 1998.

[26] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *STC 2009: Proceedings of the 4th ACM Workshop on Scalable Trusted Computing*, pages 49–54, 2009. https://doi.org/10.1145/1655108.1655117.

[27] Brooks Davis, Robert N.M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos,

J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, and Jonathan Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *ASPLOS 2019: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 379–393, 2019. https://doi.org/10.1145/3297858.3304042.

[28] Peter J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8 (4):359–389, 1976. https://doi.org/10.1145/356678.356680.

[29] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966. https://doi.org/10.1145/365230.365252.

[30] Department for Business, Energy & Industrial Strategy, Innovate UK, UK Research and Innovation, The Rt Hon Greg Clark MP, and Margot James. 'Designing out' cyber threats to businesses and personal data. https://www.gov.uk/government/news/designing-out-cyber-threats-to-businesses-and-personal-data, January 2019. Press release.

[31] Department for Business, Energy & Industrial Strategy, UK Research and Innovation, and The Rt Hon Greg Clark MP. Global businesses – including Google and Microsoft – back UK to block cyber threats with new tech. https://www.gov.uk/government/news/global-businesses-including-google-and-microsoft-back-uk-to-block-cyber-threats-with-new-tech, July 2019. Press release.

[32] Solar Designer. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63, 1997. Bugtraq mailing list.

[33] Joe Devietti, Colin Blundell, Milo M.K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008. https://doi.org/10.1145/1346281.1346295.

[34] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *EuroS&P 2016: IEEE European Symposium on Security and Privacy*, pages 147–162, 2016. https://doi.org/10.1109/EuroSP.2016.22.

[35] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr, Benjamin C. Pierce, and André DeHon. PUMP:

a programmable unit for metadata processing. In *HASP 2014: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, pages 8:1–8:8, 2014. https://doi.org/10.1145/2611765.2611773.

[36] Udit Dhawan, Cătălin Hriţcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr, Benjamin C. Pierce, and André DeHon. Architectural support for software-defined metadata processing. In *ASPLOS 2015: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2015. https://doi.org/10.1145/2694344.2694383.

[37] Michael Scott Doerrie. *Confidence in Confinement: An Axiom-free, Mechanized Verification of Confinement in Capability-based Systems*. PhD thesis, Johns Hopkins University, 2015.

[38] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *WOOT 2010: 4th USENIX Workshop on Offensive Technologies*, 2010.

[39] Tyler Durden. Bypassing PaX ASLR protection. *Phrack magazine*, 11(59):9, 2002.

[40] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *IMC 2014: Proceedings of the 2014 Internet Measurement Conference*, pages 475–488, 2014. https://doi.org/10.1145/2663716.2663755.

[41] Akram El-Korashy. A formal model for capability machines: An illustrative study towards secure compilation to CHERI. Master's thesis, Max-Planck Institute for Software Systems, 2016.

[42] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. CapablePtrs: Securely compiling partial programs using the pointers-as-capabilities principle. In *CSF 2021: 34th Symposium on Computer Security Foundations*, pages 1–16, 2021. https://doi.org/10.1109/CSF51468.2021.00036.

[43] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *SecDev 2018: IEEE Cybersecurity Development*, pages 53–60, 2018. https://doi.org/10.1109/SecDev.2018.00015.

[44] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *SIGOPS 2013: Proceedings of the*

24th ACM Symposium on Operating Systems Principles, pages 133–150, 2013. https://doi.org/10.1145/2517349.2522720.

[45] D.M. England. Capability concept mechanism and structure in System 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 63–82, 1974.

[46] Robert S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7): 403–412, 1974. https://doi.org/10.1145/361011.361070.

[47] Sharifah Yaqoub A. Fayi. What Petya/NotPetya ransomware is and what its remidiations are. In *Information Technology – New Generations*, pages 93–100. Springer International Publishing, 2018. ISBN 978-3-319-77028-4. https://doi.org/10.1007/978-3-319-77028-4_15.

[48] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. Hyperflow: A processor architecture for nonmalleable, timing-safe information-flow security. In *CCS 2018: Proceedings of the 25th Conference on Computer and Communications Security*, 2018. https://doi.org/10.1145/3243734.3243743.

[49] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. In *S&P 2020: IEEE Symposium on Security and Privacy*, pages 608–625, 2020. https://doi.org/10.1109/SP40000.2020.00098.

[50] The Jargon File. Nasal demons. http://catb.org/jargon/html/N/nasal-demons.html, 1992.

[51] Robert W. Floyd. Assigning meanings to programs. In *Mathematical aspects of computer science*, pages 19–32, 1967.

[52] John Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, 1961. https://doi.org/10.1145/366786.366800.

[53] Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *ITP 2010: First International Conference on Interactive Theorem Proving*, pages 243–258, 2010. https://doi.org/10.1007/978-3-642-14052-5_18.

[54] Anthony Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In *CPP 2017: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 125–137, 2017. https://doi.org/10.1145/3018610.3018621.

[55] Anthony C.J. Fox. Directions in ISA specification. In *ITP 2012: Third International Conference on Interactive Theorem Proving*, pages 338–344, 2012. https://doi.org/10.1007/978-3-642-32347-8_23.

[56] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. In *POPL 2021: Proceedings of 48th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2021. https://doi.org/10.1145/3434287.

[57] Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, The University of Texas at Austin, 2016.

[58] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanised logic of computation*. Springer-Verlag, 1979. https://doi.org/10.1007/3-540-09724-4.

[59] M.J.C. Gordon and A.M. Pitts. The HOL logic and system. In *Real-Time Safety Critical Systems*, volume 2, pages 49–70. Elsevier, 1994. https://doi.org/10.1016/B978-0-444-89901-9.50012-4.

[60] Richard Grisenthwaite (SVP, Chief Architect & Fellow, Arm). A safer digital future, by design. https://www.arm.com/blogs/blueprint/digital-security-by-design, October 2019.

[61] Fritz-Rudolf Güntsch. *Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation*. PhD thesis, Technische Universität Berlin, 1956. Original title: Logischer Entwurf eines digitalen Rechnergerätes mit mehreren asynchron laufenden Trommeln und automatischem Schnellspeicherbetrieb.

[62] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. N2577: A provenance-aware memory object model for C. Draft Technical Specification. ISO/IEC JTC1/SC22/WG14 N2577 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2577.pdf, 2020.

[63] Norm Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988. https://doi.org/10.1145/54289.871709.

[64] Gernot Heiser and Ben Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *ApSys 2010: Proceedings of the first ACM SIGCOMM Asia-Pacific Workshop on Systems*, pages 19–24, 2010. https://doi.org/10.1145/1851276.1851282.

[65] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. https://doi.org/10.1145/363235.363259.

[66] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP 2013: The Second Workshop on Hardware and Architectural Support for Security and Privacy*, 11, 2013. https://doi.org/10.1145/2487726.2488370.

[67] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 341–348, 1981.

[68] Shou-Ching Hsiao and Da-Yu Kao. The static analysis of WannaCry ransomware. In *ICACT 2018: 20th International Conference on Advanced Communication Technology*, pages 153–158, 2018. https://doi.org/10.23919/ICACT.2018.8323680.

[69] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX security symposium*, pages 383–398, 2009.

[70] Warren A. Hunt, Matt Kaufmann, J. Strother Moore, and Anna Slobodova. Industrial hardware and software verification with ACL2. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375, 2017. https://doi.org/10.1098/rsta.2015.0399.

[71] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4. https://software.intel.com/en-us/articles/intel-sdm, 2019. Document number: 325462-071US.

[72] ISO. Programming languages – C++, 2017. ISO/IEC 14882:2017. https://www.iso.org/standard/68564.html.

[73] ISO. Programming languages – C, 2018. ISO/IEC 9899:2018. https://www.iso.org/standard/74528.html.

[74] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In Carla Schlatter Ellis, editor, *Proceedings Of The General Track: 2002 USENIX Annual Technical Conference*, pages 275–288, 2002. ISBN 1880446006.

[75] Anita K. Jones, Robert J. Chansler Jr, Ivor Durham, Karsten Schwans, and Steven R. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *SOSP 1979: Proceedings of the seventh ACM Symposium on Operating Systems Principles*, pages 117–127, 1979. https://doi.org/10.1145/800215.806579.

[76] Tom Kilburn, David B.G. Edwards, Michael J. Lanigan, and Frank H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, 11(2):223–235, 1962. https://doi.org/10.1109/TEC.1962.5219356.

[77] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP 2009: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, 2009. https://doi.org/10.1145/1629575.1629596.

[78] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *TOCS: ACM Transactions on Computer Systems*, 32(1):1–70, 2014. https://doi.org/10.1145/2560537.

[79] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005.

[80] Neelakantan Krishnaswami. Separation logic for a higher-order typed language. In *SPACE 2006: Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, volume 6, pages 73–82, 2006.

[81] L3. The L3 model of CHERI-MIPS. https://github.com/acjf3/l3mips.

[82] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011. https://doi.org/10.1109/MSP.2011.67.

[83] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004: Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pages 75–88, 2004. https://doi.org/10.1109/CGO.2004.1281665.

[84] Henry M. Levy. *Capability-based computer systems*. Digital Equipment Corporation, 1984. ISBN 148310740X.

[85] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with return-less kernels. In *EuroSys 2010: Proceedings of the 5th European Conference on Computer Systems*, pages 195–208, 2010. https://doi.org/10.1145/1755913.1755934.

[86] William Lonergan and Paul King. Design of the B5000 system. *Datamation*, 7(5): 28–32, 1961.

[87] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, 2016. https://doi.org/10.1007/s10817-015-9360-2.

[88] Nicholas D. Matsakis and Felix S. Klock II. The Rust language. In *HILT 2014: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, volume 34, pages 103–104, 2014. https://doi.org/10.1145/2663171.2663188.

[89] Alastair J.W. Mayer. The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Computer Architecture News*, 10(4):3–10, 1982. https://doi.org/10.1145/641542.641543.

[90] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. *The design and implementation of the FreeBSD operating system*. Addison-Wesley, 2014. ISBN 9780133761825.

[91] Kayvan Memarian, Victor B.F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N.M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. In *POPL 2019: Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019. https://doi.org/10.1145/3290380.

[92] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL, February 2019. Microsoft Security Response Center.

[93] MIPS Architecture for programmers 1a. MIPS® architecture for programmers volume I-A: Introduction to the MIPS64® architecture, 2014. Imagination Technologies LTD, MD00083.

[94] MIPS Architecture for programmers 2a. MIPS® architecture for programmers volume II-A: The MIPS64® instruction set reference manual, 2016. Imagination Technologies LTD, MD00087.

[95] MIPS Architecture for programmers 3. MIPS® architecture for programmers volume III: MIPS64®/ microMIPS64™ privileged resource architecture, 2015. Imagination Technologies LTD, MD00091.

[96] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. *IEEE Security & Privacy*, 1(4):33–39, 2003. https://doi.org/10.1109/MSECP.2003.1219056.

[97] F. Lockwood Morris and Cliff B. Jones. An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, 1984. https://doi.org/10.1109/MAHC.1984.10017.

[98] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008: International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. https://doi.org/10.1007/978-3-540-78800-3_24.

[99] Randall Munroe. Heartbleed explanation. https://xkcd.com/1354/, 2014.

[100] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From general purpose to a proof of information flow enforcement. In *S&P 2013: IEEE Symposium on Security and Privacy*, pages 415–429, 2013. https://doi.org/10.1109/SP.2013.35.

[101] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Stephan A. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009. https://doi.org/10.1145/1542476.1542504.

[102] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *ISMM 2010: Proceedings of the 9th International Symposium on Memory Management*, pages 31–40, 2010. https://doi.org/10.1145/1806651.1806657.

[103] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP 2006: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, 2006. https://doi.org/10.1145/1159803.1159812.

[104] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *TPHOLs 1998: 11th International Conference on Theorem Proving in Higher Order Logics*, pages 349–366, 1998. https://doi.org/10.1007/BFb0055146.

[105] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002. https://doi.org/10.1145/503272.503286.

[106] Roger M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *SOSP 1977: Proceedings of the Sixth Symposium on Operating System Principles*, pages 1–10, 1977. https://doi.org/10.1145/800214.806541.

[107] Peter G. Neumann. Fundamental trustworthiness principles in CHERI. In *New Solutions for Cybersecurity*, chapter 6, pages 199–236. MIT Press, 2018. ISBN 9780262535373.

[108] Kyndylan Nienhuis. Verified security properties for CHERI-MIPS (Isabelle source files). https://github.com/CTSRD-CHERI/l3-cheri-mips-proofs/.

[109] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *S&P 2020: IEEE Symposium on Security and Privacy*, pages 1003–1020, 2020. https://doi.org/10.1109/SP40000.2020.00055.

[110] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *MEMOCODE 2004: Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 69–70, 2004. https://doi.org/10.1109/MEMCOD.2004.1459818.

[111] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2012. ISBN 3-540-43376-7. https://doi.org/10.1007/3-540-45949-9.

[112] Aleph One (Elias Levy). Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14, 1996.

[113] OpenSSL Project. OpenSSL project. https://openssl.org/.

[114] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 443–457, 1981.

[115] Nicole Perlroth. Yahoo says hackers stole data on 500 million users in 2014. *New York Times*, September 2016.

[116] Nicole Perlroth and David E. Sanger. Hackers hit dozens of countries exploiting stolen N.S.A. tool. *New York Times*, May 2017.

[117] Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT 1995: 6th International on Theory and Practice of Software Development*, pages 97–107, 1995. https://doi.org/10.1007/3-540-59293-8_189.

[118] Vineet Rajani, Deepak Garg, and Tamara Rezk. On access control, capabilities, their equivalence, and confused deputy attacks. In *CSF 2016: IEEE 29th Computer Security Foundations Symposium*, pages 150–163, 2016. https://doi.org/10.1109/CSF.2016.18.

[119] Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *FMCAD 2016: Formal Methods in Computer-Aided Design*, pages 161–168, 2016. https://doi.org/10.1109/FMCAD.2016.7886675.

[120] Alastair Reid. Who guards the guards? formal validation of the Arm v8-M architecture specification. In *OOPSLA 2017: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 88, 2017. https://doi.org/10.1145/3133912.

[121] Alastair Reid. *Defining interfaces between hardware and software: Quality and performance*. PhD thesis, School of Computing Science, University of Glasgow, 2019.

[122] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of ARM processors with ISA-Formal. In *CAV 2016: 28th International Conference on Computer Aided Verification*, pages 42–58, 2016. https://doi.org/10.1007/978-3-319-41540-6_3.

[123] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *EuroSys 2009: Proceedings of the 4th ACM European Conference on Computer systems*, pages 219–232, 2009. https://doi.org/10.1145/1519065.1519090.

[124] Alexander Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. PhD thesis, University of Cambridge, 2019. https://doi.org/10.17863/CAM.54548.

[125] Gerardo Richarte et al. Four different tricks to bypass StackShield and StackGuard protection. *World Wide Web*, 1, 2002.

[126] Dennis M. Ritchie. The development of the C language. In *HOPL-II: History of Programming Languages Conference*, pages 201–208, 1993. https://doi.org/10.1145/154766.155580.

[127] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *TISSEC: ACM Transactions on Information and System Security*, 15(1):2:1–2:34, 2012. https://doi.org/10.1145/2133375.2133377.

[128] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *ACSAC 2009: Twenty-Fifth Annual Computer Security Applications Conference*, pages 60–69, 2009. https://doi.org/10.1109/ACSAC.2009.16.

[129] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 44(10): 1223–1235, 1995. https://doi.org/10.1109/12.467697.

[130] Sail. The Sail ISA specification language. https://github.com/rems-project/sail.

[131] Sail CHERI-MIPS. The Sail model of CHERI-MIPS. https://github.com/CTSRD-CHERI/sail-cheri-mips.

[132] Sail RISC-V. The Sail model of CHERI-RISC-V. https://github.com/CTSRD-CHERI/sail-cheri-riscv.

[133] J.H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974. https://doi.org/10.1145/361011.361067.

[134] Oliver Schwarz and Mads Dam. Automatic derivation of platform noninterference properties. In *SEFM 2016: 14th International Conference on Software Engineering and Formal Methods*, pages 27–44, 2016. https://doi.org/10.1007/978-3-319-41591-8_3.

[135] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP 2011: Second International Conference on Interactive Theorem Proving*, pages 325–340, 2011. https://doi.org/10.1007/978-3-642-22863-6_24.

[136] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI 2013: Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 471–482, 2013. https://doi.org/10.1145/2491956.2462183.

[137] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS 2007: Proceedings of the 2007 ACM Conference on Computer and Communications Security*, pages 552–561, 2007. https://doi.org/10.1145/1315245.1315313.

[138] Scott Shane, Nicole Perlroth, and David E. Sanger. Security breach and spilled secrets have shaken the NSA to its core. *New York Times*, November 2017.

[139] Jonathan S. Shapiro and Sam Weber. Verifying the EROS confinement mechanism. In *S&P 2000: Proceeding 2000 IEEE Symposium on Security and Privacy*, pages 166–176, 2000. https://doi.org/10.1109/SECPRI.2000.848454.

[140] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *SOSP 1999: Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999. https://doi.org/10.1145/319151.319163.

[141] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. https://doi.org/10.1038/nature16961.

[142] Jessica Silver-Greenberg, Matthew Goldstein, and Nicole Perlroth. JPMorgan Chase hacking affects 76 million households. *New York Times*, October 2014.

[143] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *CCS 2015: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184, 2015. https://doi.org/10.1145/2810103.2813608.

[144] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities - provably safe stack and return pointer management. In *ESOP 2018: 27th European Symposium on Programming*, pages 475–501, 2018. https://doi.org/10.1007/978-3-319-89884-1_17.

[145] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. In *POPL 2019: The SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–28, 2019. https://doi.org/10.1145/3290332.

[146] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *S&P 2013: IEEE Symposium on Security and Privacy*, pages 574–588, 2013. https://doi.org/10.1109/SP.2013.45.

[147] Lawrence Snyder. Formal models of capability-based protection systems. *IEEE Transactions on Computers*, 30(3):172–181, 1981. https://doi.org/10.1109/TC.1981.1675753.

[148] Eugene H. Spafford. The internet worm incident. In *ESEC 1989: Second European Software Engineering Conference*, pages 446–468, 1989. https://doi.org/10.1007/3-540-51635-2_54.

[149] Eugene H. Spafford. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, 1989. https://doi.org/10.1145/66093.66095.

[150] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. In *OOPSLA 2017: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 89:1–89:26, 2017. https://doi.org/10.1145/3133913.

[151] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *S&P 2013: IEEE Symposium on Security and Privacy*, pages 48–62, 2013. https://doi.org/10.1109/SP.2013.13.

[152] The PaX team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2003.

[153] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University of Cambridge, Mathematical Laboratory, 1949.

[154] Paul Tyner. *iPAX 432 General Data Processor Architecture Reference Manual*. Intel Corporation, 1981.

[155] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. https://doi.org/10.1145/3341688.

[156] Vendicator. Stack Shield: A stack smashing technique protection tool for Linux, 2000.

[157] Robert N.M. Watson, Simon W. Moore, Peter Sewell, and Peter Neumann. CHERI research project page. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[158] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *S&P 2015: IEEE Symposium on Security and Privacy*, pages 20–37, 2015. https://doi.org/10.1109/SP.2015.9.

[159] Robert N.M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI instruction-set architecture (version 5). Technical Report UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, 2016. https://doi.org/10.48456/tr-891.

[160] Robert N.M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016. https://doi.org/10.1109/MM.2016.84.

[161] Robert N.M. Watson, Peter G Neumann, and Simon Moore. Balancing disruption and deployability in the CHERI instruction-set architecture (ISA). In *New Solutions for Cybersecurity*. MIT Press, 2017.

[162] Robert N.M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, Stacey

Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI instruction-set architecture (version 6). Technical Report UCAM-CL-TR-907, University of Cambridge, Computer Laboratory, 2017. https://doi.org/10.48456/tr-907.

[163] Robert N.M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, and Peter G. Neumann. Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre attacks. Technical Report UCAM-CL-TR-916, University of Cambridge, Computer Laboratory, 2018. https://doi.org/10.48456/tr-916.

[164] Robert N.M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI instruction-set architecture (version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019. https://doi.org/10.48456/tr-927.

[165] 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, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions CHERI instruction-set architecture (version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, 2020. https://doi.org/10.48456/tr-951.

[166] Robert N.M. Watson, Jonathan Woodruff, Alexandre Joannou, Simon W. Moore, Peter Sewell, and Arm Limited. DSbD CHERI and Morello capability essential IP (version 1). Technical Report UCAM-CL-TR-953, University of Cambridge, Computer Laboratory, 2020. https://doi.org/10.48456/tr-953.

[167] Maurice Vincent Wilkes and Roger Michael Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier, January 1979. ISBN 0-444-00357-6.

[168] Nergal (Rafal Wojtczuk). The advanced return-into-lib(c) exploits: PaX case study. *Phrack magazine*, 11(58):4, 2001.

[169] Clifford Wolf. End-to-end formal ISA verification of RISC-V processors with riscv-formal. In 7th RISC-V Workshop Proceedings, November 2017. https://doi.org/10.5446/34941.

[170] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA 2014: Proceeding of the 41st Annual International Symposium on Computer Architecture*, pages 457–468, 2014. https://doi.org/10.1109/ISCA.2014.6853201.

[171] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert Norton, Thomas Bauereiss, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N.M. Watson, and Simon W. Moore. CHERI Concentrate: Practical compressed capabilities. *IEEE Transactions on Computers*, 68(10):1455–1469, 2019. https://doi.org/10.1109/TC.2019.2914037.

[172] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974. https://doi.org/10.1145/355616.364017.

[173] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *SRDS 2003: 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, 2003. https://doi.org/10.1109/RELDIS.2003.1238076.

[174] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *CSF 2019: 32nd IEEE Computer Security Foundations Symposium*, pages 272–287, 2019. https://doi.org/10.1109/CSF.2019.00026.