VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

SECURE CALLING CONVENTION WITH UNINITIALIZED CAPABILITIES

MASTER THESIS

Sander Huyghebaert

2019-2020

Promotor: Prof. Dr. Dominique Devriese
Supervisor: Thomas Van Strydonck
Supervisor: Dr. Steven Keuchel

Sciences and Bio-Engineering Sciences

Contents

Abstract (English)

This thesis proposes a new kind of capability: *uninitialized capabilities*, a capability that cannot read uninitialized contents from memory, the capability must first be used to write something to a memory address before it can read from that address. An ISA extension for capability machines is proposed and instantiated for the CHERI capability machine. Uninitialized capabilities are then used for a secure calling convention, based on the novel calling convention by Skorstengaard et al. [13], but with the aim to reduce some of its overhead regarding clearing the stack. I modified the LLVM assembler to support uninitialized capabilities for the CHERI-MIPS processor and used the assembler then to manually modify some assembly programs to use the calling convention I propose in the thesis. The assembly programs are then evaluated for the original calling convention and the one proposed in this thesis. Finally, an exploration of the implementation for the calling convention is given for the CLang/LLVM compiler.

Abstract (Dutch)

Deze thesis stelt een nieuwe soort capability voor: *uninitialized capabilities*, een capability die geen ongeïnitialiseerde data uit het geheugen kan lezen, de capability dient eerst gebruikt te worden om te schrijven naar een adres in het geheugen vooraleer van dit adres gelezen kan worden. Een ISA uitbreiding is voorgesteld en is verwezenlijkt voor de CHERI capability machine. Uninitialized capabilities worden dan gebruikt in een secure calling convention, gebaseerd op de calling convention van Skorstengaard et al. [13], maar met als doel om de overhead van het clearen van de stack te reduceren. De LLVM assembler is aangepast om uninitialized capabilities te ondersteunen voor de CHERI-MIPS processor en is dan gebruikt om manueel enkele assembly programma's aan te passen naar de voorgestelde calling convention. Deze assembly programma's worden geëvalueerd voor de originele calling convention en de secure calling convention, als voorgesteld in deze thesis. Als laatste geef ik een exploratie voor de implementatie van de calling convention voor de CLang/LLVM compiler.

1 Introduction

Capability machines are a special type of processors. Instead of having traditional pointers, capability machines offer capabilities to interact with memory. Capabilities are unforgeable tokens that carry authority to access memory or an object. Conceptually we can view capability machines as a set of permissions, a base address, an end address and a cursor (these denote the range of authority of the capability). While capability machines were an active topic of research decades ago [5], the research interest into capability machines has grown again by merit of a recently developed capability machine, CHERI [18]. One of the biggest interests in capability machines stems from the security primitives they offer. Because capabilities have permissions and a range of authority, they seem promising to improve the security of processors and operating systems. There have been two recent proposals for alternative calling conventions that are more secure, one using local capabilities [13] and a calling convention using linear capabilities [14]. The calling convention using local capabilities had a considerable overhead regarding stack clearing and the calling convention with linear capabilities doesn't seem trivial to implement in hardware [18, Appendix D.10]. Local capabilities however, have been implemented in capability machines.

This thesis focuses on alleviating the stack clearing overhead by introducing a new kind of capability: *uninitialized capabilities*. Uninitialized capabilities prohibit the holder from reading contents from memory to which it has not first written, a only-read-after-write semantics. Having the stack capability be a local and uninitialized capability would still give it the advantages of the calling convention with local capabilities and it would get rid of the need to clear the unused part of the stack when an invoked function returns. An adversary (an untrusted piece of code trying to perform malicious actions) can save a capability in its stack frame and return control to its caller, but because the stack capability is uninitialized, when another adversary is called and tries to access that capability, it can only do so after overwriting it with something else and is thus unable to access that capability.

The first contribution of this thesis is the semantics of uninitialized capabilities (Section 4). Then an implementation of uninitialized capabilities is given for CHERI-MIPS in software (using a simulator). The modifications made to existing instructions (store instructions, load instructions and instructions modifying the cursor of a capability) are explained and new instructions are described in detail. These new instructions entail separate store instructions for uninitialized capabilities, an instruction to make a capability uninitialized and an instruction that can shrink the range of authority of a capability (similar to an already existing instruction). These instructions have been tested and work as intended.

The thesis then focuses on describing a calling convention with both local and uninitialized capabilities (Section 5). The calling convention is explained and assembly code is given to show how the different parts of the calling convention can be achieved in assembly code for CHERI-MIPS. The modifications to the assembler of the LLVM project are described (Section 6), which is used later on in the thesis to evaluate the modified and new instructions for their intended behavior and to test that the new calling convention preserves the semantics of the original program (written in C), in Section 7. The tests for the calling convention consist of two assembly files, one using the original calling convention and one in which the calling convention is modified to be the one described in the thesis (using local and uninitialized capabilities).

An evaluation is performed to quantify the difference in the number of instructions when using the original calling convention and the calling convention described in this thesis. This resulted in the observation that using the secure calling convention doubles the number of instructions. The increase in instructions depends on the size of the stack frames, which need to be cleared before a function returns to its caller and larger stack frames result in more clearing, i.e. more instructions to clear the stack. The execution time of the original calling convention and secure

calling convention is compared too, showing that there is an expected overhead due to the clearing requirements of the secure calling convention. This overhead depends on the stack frame sizes of functions, the bigger the stack frame, the more clearing is required before returning the caller of the function (functions are required to clear their own used stack frame before returning).

After the evaluation, Section 8, is devoted to discussing the implementation of the calling convention in the CLang/LLVM compiler. This exploration outlines the different parts of the compiler that need to be modified to use the new calling convention. This was not implemented due to time restrictions but the exploration will surely be useful for future work that implements this calling convention or when a new calling convention needs to be added in the CLang/LLVM compiler.

The calling convention is successfully implemented on CHERI-MIPS by manually modifying assembly programs. There is still some overhead due to clearing the stack frames of functions, but this overhead is considerably smaller than in the calling convention with only local capabilities, in which the unused part of the stack has to be cleared as well. The implementation of the calling convention in the CLang/LLVM compiler is only explored in this thesis and the actual implementation remains to be done in future work.

The results of this thesis are also part of a paper that will be submitted to POPL21[1]. This paper is the result of a collaboration between the Vrije Universiteit Brussel and Aarhus University, Denmark. The paper describes uninitialized capabilities and the calling convention on a different capability machine (simpler than CHERI and it has enter capabilities). The paper theoretically underpins the calling convention and its guarantees regarding well-bracketed control flow and local state encapsulation by proving that an implementation of the awkward example (a program that depends on well-bracketed control flow) is correct.

2 Background

This background section provides the reader with the required knowledge regarding capability machines, CHERI and the calling conventions studied as a basis for the contribution of the thesis. It is recommended to read the background, capability machines are not part of the standard knowledge of computer scientists but are the main focus point of the thesis.

2.1 Capability Machines

Capability machines are a special type of processor that replaces pointers with capabilities. Conceptually, capabilities are tokens that carry authority to access memory or an object. When capabilities represent software defined authority like invoking objects or closures, they are referred to as *object capabilities*. This thesis will focus on primitive capabilities for accessing memory. The permissions to access memory can be read only, read and write to, execute, ... The idea of capabilities was first formally defined by Dennis and Van Horn [2] and has been further explored in the decades after.

The first capability machine dates back from 1959 with the *Rice University Computer* and the development and research interest of capability machines slowed down significantly after the *iAPX 432* from *Intel* in 1981 [5]. In 2014, researchers of the University of Cambridge developed a new capability machine: *CHERI*, on which I will provide an implementation of the contribution of this thesis to capability machines.

[1] https://popl21.sigplan.org/

It is important that capabilities cannot be forged, as forging them with certain permissions, memory bounds, etc. would defeat their purpose. One of the solutions to ensure the unforgeability of capabilities is to provide specialized instructions to work with capabilities. Capabilities might however need to be stored in primary memory or secondary memory instead of just the registers on the processor and one of the most used techniques to ensure valid capabilities is the use of tagged memory [3]. Every possible capability location will have a tag denoting if that location contains a capability or not. Capabilities for which the tag is not set cannot be used to dereference memory.

Some common permissions found on capability machines are:

- **R**: read-only;
- **RW**: read, write;
- **RX**: read, execute;
- **RWX**: read, write, execute.

Capabilities can be formally represented by a 4-tuple similar to the representation used by Skorstengaard et al. [13], (*permissions*, *base*, *end*, *cursor*), this tuple contains the *permissions* of the capability, the range to which these permissions apply *[base, end]* and a *cursor* in that range. An advantage of capabilities over pointers is that they carry permissions and bounds to access memory, whereas a pointer is just a number indicating a memory address with no restrictions, this means that capabilities require more bits than pointers. Furthermore, the semantics of the C programming language regarding pointers often specifies *undefined* behavior, for example when performing out-of-bounds pointer arithmetic [6]. The behavior of capabilities is better defined as it can be deducted from the permissions and range of authority it has. If the cursor of the capabilities goes out-of-bounds then it cannot be used to dereference that address.

The following two sections describe two kinds of capabilities that will be used later in the thesis.

2.1.1 Local Capabilities

Capabilities can be either *global* or *local*. Global capabilities correspond to the capabilities explained above and can be stored by both global and local capabilities that have the write permission. The new kind of capability is the *local* capability, which behaves different for storing local capabilities. Local capabilities can only be stored into memory by using a capability that is local itself and has the write-local permission [18, Section 2.3.10].

2.1.2 Sealed Capabilities

One more capability of interest for this thesis is the *sealed* capability. A sealed capability is immutable and cannot be used to read or write to memory. If one wants to use a sealed capability or modify it, it must first be unsealed. Unsealing a sealed capability can only be done by using the same seal by which it was sealed.

2.2 CHERI

CHERI (**C**apability **H**ardware **E**nhanced **RI**SC **I**nstructions) is an ISA extension that introduces capabilities. The main goals of CHERI are fine-grained memory protection, software compartmentalization and backwards compatibility [18].

The CHERI ISA extension proposes a 64-bit, 128-bit and 256-bit capability representation format [18], this thesis will focus on the 256-bit capability format but the ideas and concepts represented in the thesis should be applicable to the other capability formats as well.

Figure 1 shows the current 256-bit capability format.

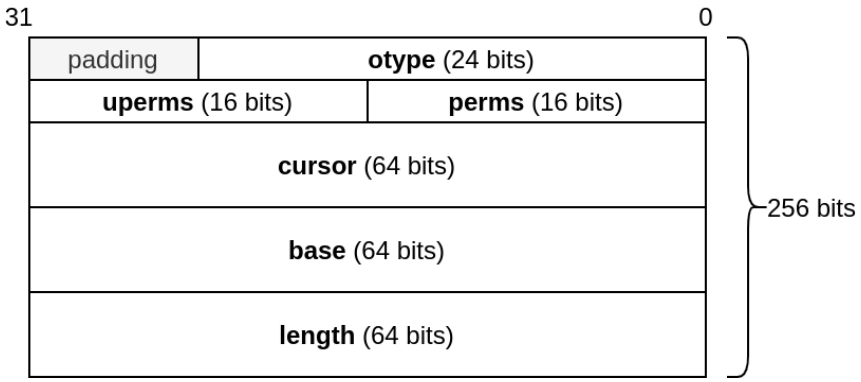


Figure 1: 256-bit Capability Representation Format

The important fields of a capability used throughout the thesis are the permissions, cursor, base and length fields. In our formal representation of capabilities we have an *end* field instead of *length* but it should be straightforward to see that $end = base + length$.

The CHERI ISA extension is implemented for the MIPS ISA (CHERI-MIPS) and has a functional hardware implementation.

2.2.1 Sealed Capabilities

The definition of sealed capabilities was given in Section 2.1.2. This section will explain how CHERI supports sealed capabilities.

Unsealed capabilities have an *otype* (see Figure 1) of -1, which is the default for all capabilities. A capability can be sealed by using the *CSeal* instruction, which is of the form: `cseal cd, cb, ct`. Capability register *cb* contains the capability to be sealed and the sealed capability will be written to *cd*. The seal itself is taken from the capability residing in *ct* (which needs to have the `Permit_Seal` permission) more specifically, the 24 least significant bits of the cursor of *ct*. This cursor is not allowed to be greater than the maximum otype (which is $2^{24} - 1 - 16$, the -16 subtracts the reserved otypes that have a special meaning and cannot be used as a normal seal). The cursor itself also needs to be in the bounds of the capability used for sealing (*ct*). If these conditions are not met than an exception is raised (some other conditions apply too but are irrelevant for the discussion here). The *otype* of a capability is transparent and can be requested by using the *CGetType* instruction [18, Section 8.10].

I will discuss two options to unseal a capability. The first option to unseal a sealed capability is by using the *CUnseal* instruction, of the format `cunseal cd, cb, ct` which is similar to the CSeal instruction (*ct* needs to have the `Permit_Unseal` permission). The second option, the one that will be used later in the thesis, is using the *CCall* instruction. CCall is of the format `ccall cs, cb, selector` (selector is an 11-bits constant), CHERI currently supports two selectors, the only selector used throughout the thesis is $selector = 1$, for more details about CCall using $selector = 0$, refer to the CHERI technical report [18, page 181]. Both capabilities in registers *cs* and *cb* will be unsealed, but only if they **both** have the same seal. Furthermore, *cs* should be a capability with the execute permission and *cb* should not have the execute permission set.

The contents of *cs* will be placed in the program counter capability register (PCC) and *cb* in the invoked data capability register (IDC).

CCall thus takes a pair of sealed capabilities, a code capability and a data capability and will unseal the pair and place the code capability in the PCC (i.e. jumping to it) and places the data capability in the IDC.

2.3 Calling Convention

This section will describe what a calling convention is. The next 2 sections discuss 2 specific calling conventions, the calling convention being used on CHERI-MIPS and a secure calling convention on a theoretical capability machine.

A calling convention is a set of guidelines for [15]:

- how to use the registers available on a processor;
  - how many arguments are passed in registers and which registers are used for the arguments;
  - in which register(s) the return value(s) will reside;
- what needs to happen before calling a function;
- what needs to happen after a function returns to its caller;
- the prologue of a function.
- the epilogue of a function;

This definition of calling conventions will be used throughout the thesis to describe calling conventions.

2.4 CHERI-MIPS Calling Convention

This section describes the calling convention used in CHERI-MIPS (for the purecap application binary interface (ABI), which describes the used calling convention, register conventions, etc.), which is similar to the MIPS-N32/N64 calling convention[18, Section 4.11]. The CHERI ISAv7 technical report mentions the capability register conventions used in the calling convention in Section 4.11[18]. The remaining information for the CHERI-MIPS calling convention can be found by consulting the CHERI programmer's guide [19, Section 6.2], the MIPSPro™ N32 ABI Handbook[11], the MIPS Assembly Language Programmer's Guide[15] and the, for CHERI modified, LLVM compiler[2] in the *MIPS* backend. It should be noted that the calling convention will be described without using the frame pointer. The general-purpose registers (GPR) are denoted as `$<num>` and capability registers as `$c<num>`.

2.4.1 Invoking A Function

Invoking a function happens by jumping to it, using the *CJALR* (capability jump and link register), which takes 2 operands, the capability register containing the capability to jump to and a capability register to link the return (which will be a capability for the current PC + 8).

Before being able to invoke a function it is necessary to create a capability that points to the code of the function we want to call. The first thing to do is get a PC-relative address for the Global Offset Table (GOT), this is done by using the following instruction sequence:

[2] https://github.com/CTSRD-CHERI/llvm-project

```
1 lui $1, %pcrel_hi(_CHERI_CAPABILITY_TABLE_-8)
2 daddiu $1, $1, %pcrel_lo(_CHERI_CAPABILITY_TABLE_-4)
3 cgetpccincoffset $c1, $1
```

All that's left now is to create an entry point capability for the function we need:

```
1 # replace fn_symbol_name with the name of the function to jump to
2 clcbi $c12, %capcall20(fn_symbol_name)($c1)
```

Before we jump to the capability for the function we want to call, any caller-saved registers that we use need to be pushed on the stack.

In CHERI-MIPS $c12 is used as the entry point capability and $c17 as the return point capability, consequently jumping to a function will always look like `cjalr $c12, $c17`.

When the function returns we need to restore the caller-saved registers. The caller-save registers are *$c11-16, $c25*.

2.4.2 Argument Passing

Argument passing follows the MIPS calling convention rules, in which the first eight integer arguments are passed in the registers $4-11. Additionally, the first eight capability arguments are passed in registers $c3-10. All other arguments are passed on the stack.

2.4.3 Returning From A Function

In order to return from a function a jump instruction is used with as operand a capability to return to. For CHERI-MIPS this is the *CJR* instruction (jump capability register) and by convention the capability to return to is in capability register 17 (*$c17*), i.e. the full instruction to return to the caller looks like `cjr $c17`.

2.4.4 Function Prologue

The function prologue describes what needs to happen at the beginning of a function.

The following steps describe the function prologue for CHERI-MIPS:

- Adjust the stack capability by decreasing the offset with the size of the stack frame (i.e. push stack frame);
- Spill the callee saved registers onto the stack.

The callee saved registers are: *$16-23, $28, $30-31, $c17-25*.

2.4.5 Function Epilogue

The function epilogue describes what needs to happen at the end of a function. The epilogue undoes the setup of the stack frame by the prologue:

- Increase the offset of the stack capability with the stack frame size (i.e. popping the stack frame);
- Restoring the callee saved registers.

2.5 Calling Convention Using Local Capabilities

The calling convention described in this section is the one defined by Skorstengaard et al.[13] It makes use of local capabilities, which are capabilities that can only be kept in registers except for when you have a capability with the write-local permission. This requires that capabilities have a new permission added, the **L** (Local) permission and the possible combinations for this permission are **RWL** and **RWLX**.

An important requirement for this calling convention is that the stack capability is a local capability with the **RWLX** permission.

The calling convention aims to have the well-bracketed control flow and local state encapsulation properties. These are defined as:

> Well-bracketed control flow (WBCF) expresses that invoked functions must either return to their callers, invoke other functions themselves or diverge, and generally holds in programming languages that do not offer a primitive form of continuations [14].
>
> Local state encapsulation (LSE) is the guarantee that when a function invokes another function, its local variables (saved on its stack frame) will not be read or modified until the invoked function returns [14].

2.5.1 Invoking A Function

The first thing to do is to save the used caller-saved registers on the stack.

As a return capability for the callee we need to create an enter capability (a permissions that cannot be used to read, write or execute and a capability with this permissions cannot be modified but they can be jumped to and when jumped to their permission becomes **RX**). To create this enter capability we will store the restore instructions on the stack and after these instructions our return capability and stack capability. We then create the local enter capability for the restore instructions, return and stack capability. The restore instructions will restore the return and stack capability. Figure 2 shows what this would look like on the stack.

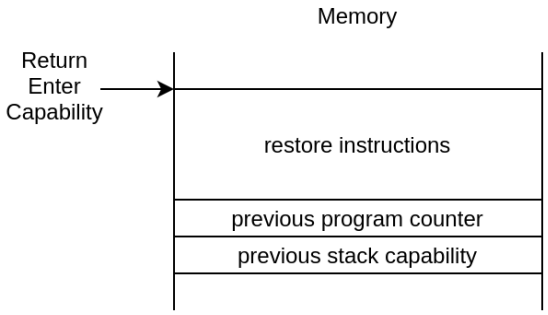


Figure 2: Stack frame with restore instructions

Now we can also restrict the stack capability to only cover the unused part of the stack. We need to clear this unused part because it may contain data and capabilities left there by previously executed code. Any non-argument registers are also cleared.

After this setup is done a jump can be made to the function. A check is required if the jump is done using a function pointer (for example callbacks), to see if that function pointer is global (i.e. not local). If this check fails then the function invocation fails.

When the callee returns using the given enter return capability it will execute the restore instructions, after which we can remove the return and stack capability from the stack, as well as the restore instructions. All that's left is to restore our caller-saved registers from the stack.

2.5.2 Argument Passing

The calling convention is similar to standard C calling conventions, so it is safe to assume that much like other calling conventions the first x arguments are passed in registers and the remaining arguments are passed on the stack. This part of the calling convention depends on the processor on which it is used.

2.5.3 Returning From A Function

Returning from a function happens by jumping to the return capability. This return capability is an enter capability that will restore the stack and return capability for the caller and continue by restoring caller-saved registers.

2.5.4 Function Prologue

The callee needs to check that the stack capability is **RWLX**. The necessary stack setup happens by the caller instead of by the callee. There are no more callee-saved registers, because all non-argument registers are cleared, if the caller still needs the contents of some non-argument register than the caller should save them, i.e. all registers can be regarded as caller-saved registers.

2.5.5 Function Epilogue

The following steps should be taken before jumping to the return capability:

- clear the non-return value registers;
- clear the used part of the stack (spilled registers, local state from invoking other functions, ...);
- clear the unused part of the stack.

The unused part of the stack needs to be cleared because if this function invoked adversary code it could be possible that data has been stored in the unused part of the stack. This data could be an old stack capability or return capability which could then be used later on, for example to get access to a part of the stack it should not be able to access (in the case of storing an old stack capability).

3 Related Work

Capability machines are a special type of processor that offer *capabilities*, which can be described as unforgeable tokens that carry authority. The first capability machines can be traced back to 1959 with the Rice University Computer. Capability machines have seen active research interest for a few decades but became less active after the Intel iAPX 432 from 1981 [5].

Capabilities were originally formally defined by Van Horn and Dennis [2] and after this formalization the term capability found more widespread use. An important addition to capability systems was the concept of sealing [7, 12] which makes it impossible to modify or use the capability without unsealing it first.

In Fabry's paper from 1974 [3] he compares capability addressing with segmented primary memory addressing and predicted that the use of tags to identify capabilities would dominate.

One of the most prominent recent capability machines, from 2014, is CHERI [18], which is an Instruction Set Architecture (ISA) extension. CHERI was strongly influenced by the M-Machine [1], which uses tagged memory. The tagged memory implementation of CHERI itself is quite efficient with less than 5% overhead for most applications [4]. The team behind CHERI also uses different systems engineering methods, the traditional methods are used but the CHERI team is also using rigorous methods to develop a more security-enhanced processor and to write mechanized proofs that the intended security properties of the capability machine hold [8].

CHERI aims to provide fine-grained memory protection, highly scalable software compartmentalization and backwards compatibility. A prototype CHERI-MIPS processor has been developed and proposals have been made to add the CHERI ISA extension for the x86-64 ISA (CHERI-x86-64) and RISC-V ISA (CHERI-RISC-V). This capability machine uses tagging to identify capabilities by having a tag for each capability-sized, capability-aligned word in primary memory. The software stack of CHERI consists of CLang/LLVM, CHERIBSD and a QEmu emulator (as well as some ported software like nginx, sqlite, ...) and the development of this software stack started in April 2014 [17]. CHERI provides a 256-bit capability format and a compressed format, called CHERI concentrate [22], which is the successor for the deprecated CHERI-128 and CHERI-64 compression schemes [18]. CHERI concentrate offers 128-bit and 64-bit capability formats.

CHERI offers a scalable software compartmentalization API by using a central trusted stack and a stack for each created compartment [20]. Having a stack for each compartment is a considerable memory overhead and makes supporting higher-order settings more difficult (i.e. closures, objects). One of the challenges of scalable software compartmentalization is finding a way to automate the compartmentalization of programs, with recent work for CHERI software compartmentalization automation by Tsampas et al [16].

Capabilities can offer more security than normal pointers and they are being used in formal approaches for secure compilation [9]. Secure calling conventions for capability machines have been researched by Skorstengaard et al. and their first approach uses local capabilities similar to those of CHERI [13], this calling convention uses a single stack instead of the compartmentalization available in CHERI. A disadvantage of that calling convention is that it requires register clearing and clearing the entire stack, which is a considerable overhead. CHERI does offer a more efficient instruction for clearing up to 16 registers, *CClearRegs* [21], it does not have a big enough impact to reduce the overhead of zeroing registers. An alternative calling convention could avoid the requirement to clear the entire stack by only requiring that the used stack frame must be cleared. This reduces the overhead of clearing the stack and should be an acceptable overhead. An adjustment to the calling convention using local capabilities has been made to take this into account, this modified calling convention uses linear capabilities, capabilities that cannot be duplicated and would avoid the need of clearing the entire stack [14]. Linear capabilities

have not been added to CHERI as it is not trivial to implement them in hardware (see appendix D.10 in the CHERI ISA v7 technical report [18]), which means that the stack clearing problem has remained unsolved.

4 Uninitialized Capabilities

Uninitialized capabilities are a new type of capabilities. They are memory capabilities which represent read-write authority to a range of memory, except that they do not allow reading the initial contents of the memory. The memory first needs to be overwritten before it can be read. This type of capability requires a new permission to be added to capabilities (**U**: uninitialized) and prevents the holder of the capability from reading memory that they have not first initialized. Figure 3 clarifies this concept a bit more.

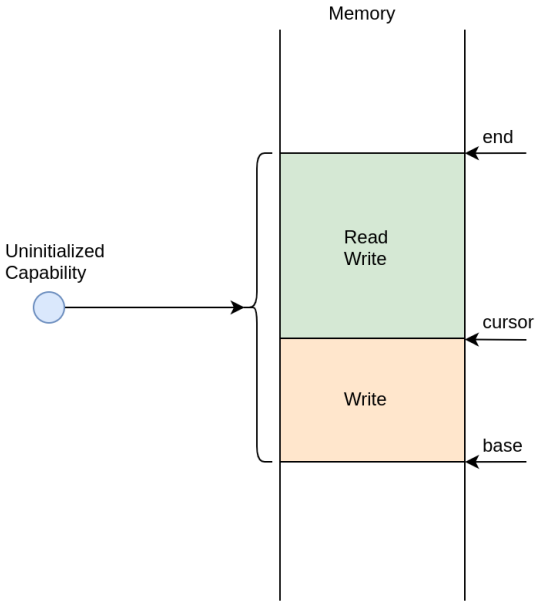


Figure 3: Uninitialized Capabilities Concept

Formally, uninitialized capabilities grant the following authority:

- permission to read in $[cursor, end]$;
- permission to write in $[base, end]$;
- when writing immediately below the cursor, the cursor will be decremented so that the holder of the uninitialized capability is able to read from the location it has just written to.

Uninitialized capabilities let the read range grow towards the base (after writing immediately below the cursor), this reflects the way the stack grows. For MIPS, the stack grows downwards. If the stack grows upwards, the authority of uninitialized capabilities will need to be adjusted accordingly, having the read permission in $[base, cursor]$ and have it grow towards the end (when writing immediately above the cursor).

The full set of permissions becomes (for a capability machines with local capabilities):

- **R**: read-only;
- **RW**: read, write;

- **RX**: read, execute;
- **RWX**: read, write, execute.
- **RWLX**: read, write-local, execute.
- **RWL**: read, write-local;
- **URW**: read between $[cursor, end]$, write between $[base, end]$;
- **URWL**: read between $[cursor, end]$, write-local between $[base, end]$;

Combinations of the **U** permission and **X** permission are invalid, executing an uninitialized capabilities would require incrementing the program counter (and thus the cursor of the uninitialized capability), which means that the non-readable range of the capability would grow. Another option is to allow the combination of the **U** permission with the **X** permission, but when jumping to an uninitialized capability transform it into a normal capability for the range $[cursor, end]$ before placing it in the program counter capability register.

4.1 Implementation Overview

This section gives an overview of the concrete design of uninitialized capabilities for the *CHERI* capability machine, particularly the CHERI-MIPS ISA and the 256-bit capability format. However, the general concept is not limited to CHERI-MIPS or the 256-bit capability format. The concept of uninitialized capabilities is an addition to capability machines in general, and particularly the CHERI protection model, regardless of the architecture is it run on.

4.1.1 Uninitialized Permission Bit

The first modification that needs to be made to CHERI capabilities is the addition of a new permission, the uninitialized permission. In the 256-bit capability format there are a few unused bits (padding bits) available so one of those bits is now used for the uninitialized permission, as can be seen in Figure 4.

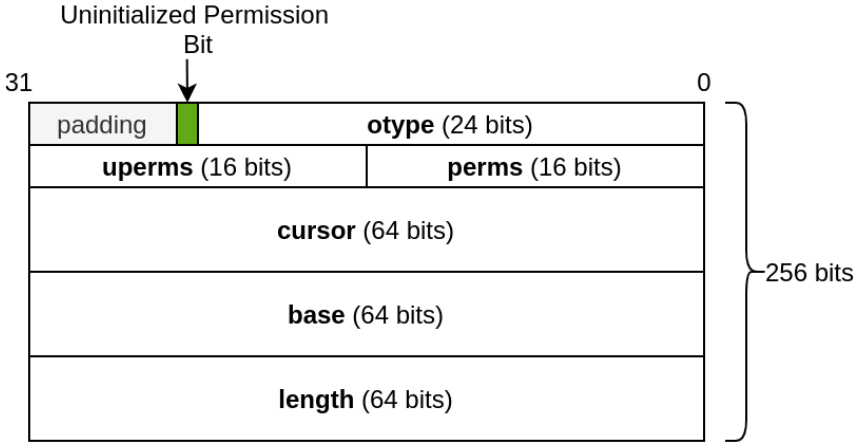


Figure 4: Modified 256-bit representation of a capability

4.1.2 Instruction Modifications

A few instruction were modified to take the uninitialized permission into account. What follows is a list of the instructions modified and a description of what that modification entails:

Load via Capability Register (CL[BHWD][U]/CLC): When load instructions are given a capability with the uninitialized permission set, it is not allowed to load from an address lower than the cursor.

Set/Increment Offset (CSetOffset/CIncOffset//CIncOffsetImmediate/CSetAddr/ CAndAddr): Instructions that modify the cursor of an uninitialized capability are not allowed to set the cursor lower than it originally was. The only way of lowering the cursor is by using the uninitialized store instructions.

4.1.3 New Instructions

A few new instructions have been added for the implementation of uninitialized capabilities:

Get Uninitialized Bit of a Capability (CGetUninit): This instructions has 2 parameters, the general-purpose register to store the uninitialized bit of the capability into and the capability of which the uninitialized bit is requested.

Uninitialize a Capability (CUninit): An instruction to make a capability uninitialized. This instruction takes a source capability register and a destination capability register that will contain the capability from the source register but with the uninitialized permission set. An error will be raised if the original capability did not have read-write authority.

The instructions to get the uninitialized bit and set it can be replaced by adding it to the existing set of permissions of capabilities. This would require a similar scheme as for local capabilities. Instead of having an uninitialized permission, the permission would indicate if a capability is initialized, i.e. capabilities would have a bit saying it is initialized if that bit is set and uninitialized if it is not set. This would allow making a capability uninitialized by using the *CAndPerm* instruction, which will set the permissions of the capability to the logical `&` (and) of the current permissions of the capability with the permissions supplied in a general-purpose as argument to *CAndPerm*. The "initialized" bit can then be set to 0 (i.e. indicating the capability is uninitialized) by having the corresponding bit in the general-purpose be zero. This implementation choice is used for local capabilities, capabilities are local if the *global* bit is unset. In this thesis the choice was made to use separate instructions for the uninitialized bit, otherwise every occurrence of the existing *CAndPerm* instructions needs to be updated to make sure that initialized capabilities remain initialized.

Drop The Uninitialized Permission of a Capability (CDropUninit): This instruction allows making an uninitialized capability normal again, i.e. dropping the uninitialized permission, once the entire range denoted by the base and end of the capability has been written to. That means that if no uninitialized contents resides in the range of an uninitialized capability, then it is safe to drop the uninitialized permission of the capability. Using this instruction an uninitialized capability for an initialized array or struct can be converted into a capability without the uninitialized bit set, which allows using negative increments for the offset (for example to loop with a pointer over an array starting from the end of the array towards the base of the array, which is not possible with uninitialized capabilities).

Uninitialized Store (UCS[BHWD]/UCSC): These instructions are modified versions of their not-uninitialized counterparts (CS[BHWD], CSC). They behave similarly to the normal store instructions, except when the given offset is -1 and the capability used for the store is uninitialized. In that case, the capability written to the destination capability register will have the cursor of the source capability decremented by the number of bytes written (i.e. 1 for a byte, 2 for a half word, 4 for a word, 8 for a double word and 32 for capabilities when using the 256-bit capability format). Specifying an offset of -1 is the **only** way to decrement the cursor. This instruction takes 4 arguments, a destination capability register (which will contain the source capability but possibly with its cursor modified if the offset was -1), a source register for the data to write, an offset and a source capability register.

The original store instructions for capabilities are **not** modified (CSC, CSW, ...), but instead I propose to add new instructions to handle the uninitialized permission. The new instructions write to a capability register the possibly modified capability (if it has the **U** permission set and the given offset is -1), while the original instructions do not write to a register but instead allow specifying a register containing another offset to be added to the cursor of the capability.

One additional instruction is required to modify the bounds of uninitialized capabilities:

Shrink a Capability (CShrink[Imm]): CShrink is an instruction with 3 parameters, the destination capability register, the source capability register and a general-purpose register (GPR), or alternatively an unsigned immediate for CShrinkImm. The capability from the source register will be modified by setting $end = cursor$ and $base = value\ in\ GPR$ for CShrink. For CShrinkImm $end = cursor$ and $base = base + immediate$. CShrink[Imm] will raise an exception if the $end < cursor$ (the original end and $cursor$ of the capability) or if $newBase < base$, these conditions prevent expanding the range of authority of the capability.

In the CHERI-MIPS ISA a similar instruction is already available, **CSetBounds**, but this instruction did not meet the needs of uninitialized capabilities. It adjusts the bounds by setting $base = cursor$ and $end = cursor + immediate$, where immediate is either the value from the general-purpose register specified in the instruction or an unsigned immediate value.

The issue with using this instruction in combination with uninitialized capabilities arises when trying to lower the end of the uninitialized capability, but maintain the same $base$. Using CSetBounds this would require first setting $cursor = base$, calculate the offset for the new end, perform the CSetBounds instruction and then setting the $cursor$ back to its value before it was set to $base$. This obviously means lowering the cursor ($cursor = base$) which is not permitted for uninitialized capabilities.

4.2 CHERI-MIPS ISA Extension

The implementation of the design of uninitialized capabilities, as described in the previous section, has been instantiated for CHERI-MIPS in software (using a simulator). Each instruction will be given with its format, encoding, purpose, description, semantics, exceptions and optional notes. This is the same template as used in the CHERI technical report [18, chapter 7]. For the modified instructions only the modifications will be given for the template and a small description, for the complete information of the instruction, please refer to the CHERI technical report [18, chapter 7].

The semantics of each instruction will be shown by its Sail code. Sail is an imperative programming language that is used to describe instruction set architectures. Sail can generate a simulator from the sail code (in OCaml or C) but can also generate code for theorem provers [18,

page 167]. Note that in the semantics section of each instruction, the common definitions are hyperlinks (in red) to the corresponding type and explanation of a function in Appendix A. It is not required to understand these definitions to understand the code, the name of each function should be descriptive enough to understand more or less what it does.

CL[BHWD][U]

Description

Load bytes via a capability. The loaded bytes can be sign extended or zero extended, for example: `clb $1, $zero, 0($c1)` will load a single byte, sign extended into register *$1* from the cursor of capability *$c1*.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let 'size = wordWidthBytes(width);
  let cursor = getCapCursor(cb_val);
  let vAddr = (cursor + unsigned(rGPR(rt)) + size*signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if cb_val.uninit & vAddr < cursor then
    raise_c2_exception(CapEx_UninitLoadViolation, cb)
  else if not (isAddressAligned(vAddr64, width)) then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let pAddr = TLBTranslate(vAddr64, LoadData);
    memResult : bits(64) = extendLoad(MEMr_wrapper(pAddr, size), signext);
    wGPR(rd) = memResult;
  }
}
```

Exceptions

On top of the existing conditions on which an exception is thrown, one additional condition is added that raises a coprocessor 2 exception:

- cb.uninit is set and $addr < cursor$.

CLC

Description

Load capability via a capability.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let cursor = getCapCursor(cb_val);
  let vAddr = (cursor + unsigned(rGPR(rt)) + 16 * signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if cb_val.uninit & vAddr < cursor then
    raise_c2_exception(CapEx_UninitLoadViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdEL, vAddr64)
  else
  {
    let (pAddr, macr) = TLBTranslateC(vAddr64, LoadData);
    let (tag, mem) =
      MEMr_tagged(pAddr, cap_size, cb_val.permit_load_cap & not (macr == Clear
          ↪ ))
    in
    if tag & macr == Trap then
      raise_c2_exception(CapEx_TLBLoadCap, cb)
    else let cap = memBitsToCapability(tag, mem) in
        writeCapReg(cd, cap);
  }
}
```

Exceptions

On top of the existing conditions on which an exception is thrown, one additional condition is added that raises a coprocessor 2 exception:

- cb.uninit is set and $addr < cursor$.

CSetOffset

Description

Set the offset of a capability, i.e. the new cursor becomes $base + offset$.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
let offset = getCapOffset(cb_val);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cb_val.tag & cb_val.uninit & (unsigned(rt_val) < offset) then
  raise_c2_exception(CapEx_UninitViolation, cb)
else
{
  let (success, newCap) = setCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, unrepCap(newCap))
}
```

Exceptions

There is one more case when an exception will be raised:

- *cb.tag* is set, *cb.uninit* is set and the value of *rt* is negative.

Notes

- It is not possible to lower the cursor by supplying a negative offset when the capability is uninitialized.

CIncOffset

Description

Increment the offset of a capability, i.e. the new cursor becomes $cursor + delta$.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed & (rt != 0b00000) then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cb_val.tag & cb_val.uninit & rt_val[63] then
  raise_c2_exception(CapEx_UninitViolation, cb)
else
{
  let (success, newCap) = incCapOffset(cb_val, rt_val);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, unrepCap(newCap))
}
```

Exceptions

There is one more case when an exception will be raised:

- $cb.tag$ is set, $cb.uninit$ is set and the value of rt is negative.

Notes

- It is not possible to lower the cursor by supplying a negative offset when the capability is uninitialized.

CIncOffsetImm

Description

Same as for CIncOffset but with an immediate.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let imm64 : bits(64) = sign_extend(imm);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cb_val.tag & cb_val.uninit & imm64[63] then
  raise_c2_exception(CapEx_UninitViolation, cb)
else
{
  let (success, newCap) = incCapOffset(cb_val, imm64);
  if success then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, unrepCap(newCap))
}
```

Exceptions

There is one more case when an exception will be raised:

- *cb.tag* is set, *cb.uninit* is set and *imm* is negative.

Notes

- It is not possible to lower the cursor by supplying a negative immediate when the capability is uninitialized.

CSetAddr

Description

Sets the address (= cursor) of a capability.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if cb_val.tag & cb_val.uninit & (rt_val <_u cb_val.address) then
  raise_c2_exception(CapEx_UninitViolation, cb)
else
{
  let (representable, newCap) = setCapAddr(cb_val, rt_val);
  if representable then
      writeCapReg(cd, newCap)
  else
      writeCapReg(cd, unrepCap(newCap));
}
```

Exceptions

There is one more case when an exception will be raised:

- *cb.tag* is set, *cb.uninit* is set and the value of *rt* is smaller than the current address of *cb* (this means lowering the address, which is not allowed for uninitialized capabilities).

Notes

- It is not possible to lower the cursor by supplying an address lower than the current address of the capability if the capability is uninitialized.

CAndAddr

Description

Masks the address of a capability, i.e. `cursor = cursor & supplied_addr`.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let rt_val = rGPR(rt);
let addr = cb_val.address;
if cb_val.tag & cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let newAddr = cb_val.address & rt_val;
  if cb_val.tag & cb_val.uninit & (newAddr <_u addr) then
    raise_c2_exception(CapEx_UninitViolation, cb)
  else {
    let (representable, newCap) = setCapAddr(cb_val, newAddr);
    if representable then
      writeCapReg(cd, newCap)
    else
      writeCapReg(cd, unrepCap(newCap));
  }
}
```

Exceptions

There is one more case when an exception will be raised:

- *cb.tag* is set, *cb.uninit* is set and *newAddr* is smaller than the current address of *cb* (this means lowering the address, which is not allowed for uninitialized capabilities).

Notes

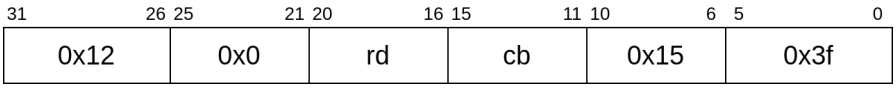
- It is not possible to lower the cursor by supplying a value in *rt* that, when performing the bitwise AND operator with the current address, would result in an address lower than the current address of the capability if the capability is uninitialized.

CGetUninit

Format

CGetUninit rd, cb

Encoding



Description

The uninitialized bit of the capability in register *cb* is written (zero extended) to *rd*.

Semantics

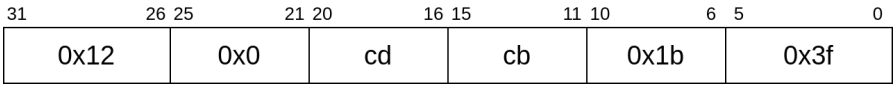
```
checkCP2usable();
let capVal = readCapReg(cb);
wGPR(rd) = zero_extend(capVal.uninit);
```

CUninit

Format

CUninit cd, cb

Encoding



Description

Capability in capability register *cb* is written to capability register *cd* but with the *uninit* bit set.

Semantics

```
checkCP2usable();
let cap = readCapReg(cb);
if cap.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else
{
  let newCap = uninitCap(cap);
  writeCapReg(cd, newCap)
}
```

Exceptions

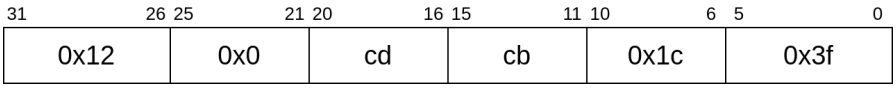
An exception is raised if the capability in *cb* is sealed.

CDropUninit

Format

CDropUninit cd, cb

Encoding



Description

The uninit permission of the uninitialized capability in capability register *cb* is dropped if the cursor of the capability is equal to the base, i.e. the uninitialized capability has no more uninitialized content in the range of memory it denotes.

Semantics

```
checkCP2usable();
let cap = readCapReg(cb);
let offset = getCapOffset(cap);
if cap.sealed then
 raise_c2_exception(CapEx_SealViolation, cb)
else if not (cap.uninit) | offset != 0 then
 raise_c2_exception(CapEx_UninitViolation, cb)
else
{
 let newCap = dropUninitFromCap(cap);
 writeCapReg(cd, newCap)
}
```

Exceptions

A coproccessor 2 exception is raised if:

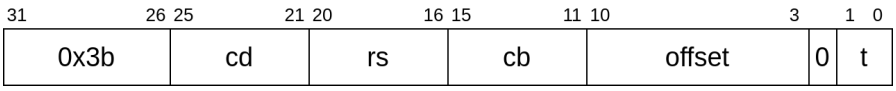
- the capability in *cb* is sealed;
- the capability in *cb* is not uninitialized;
- the cursor of the capability in *cb* is different from the base.

UCS[BHWD]

Format

UCSB cd, rs, offset(cb)
UCSH cd, rs, offset(cb)
UCSW cd, rs, offset(cb)
UCSD cd, rs, offset(cb)

Encoding



Description

This instructions stores some or all of register rs into the memory location specified by $cb.base + cb.offset + rt + 2^t * offset$ if the capability cb has the permission to store data. The t field indicates how many bits are stored to the memory location:

- **0**: byte (8 bits)
- **1**: halfword (16 bits)
- **2**: word (32 bits)
- **3**: doubleword (64 bits)

The least-significant end of the register is used when less than 64 bits need to be stored.

When the given offset equals -1, the cursor of the capability cb is decremented by the amount of bytes written to memory and the modified capability with the decremented cursor is written to cd. If the offset is not -1, cb is copied to cd.

Semantics

```
checkCP2usable();
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_store) then
  raise_c2_exception(CapEx_PermitStoreViolation, cb)
else
{
  let size = wordWidthBytes(width);
  let cursor = getCapCursor(cb_val);
  let vAddr = (cursor + size * signed(offset)) % pow2(64);
  let vAddr64= to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
```

```
    else if vAddr < getCapBase(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
    else if not (isAddressAligned(vAddr64, width)) then
      SignalExceptionBadAddr(AdES, vAddr64)
    else
    {
      let pAddr = TLBTranslate(vAddr64, StoreData);
      let rs_val = rGPR(rs);
      if cb_val.uninit & offset == ones() then {
        let decr = to_bits(64, negate(size));
        let (_, newCap) = incCapOffset(cb_val, decr);
        writeCapReg(cd, newCap)
      } else {
        writeCapReg(cd, cb_val);
      };
      match width
      {
        B => MEMw_wrapper(pAddr, 1) = rs_val[7..0],
        H => MEMw_wrapper(pAddr, 2) = rs_val[15..0],
        W => MEMw_wrapper(pAddr, 4) = rs_val[31..0],
        D => MEMw_wrapper(pAddr, 8) = rs_val
      }
    }
  }
```

Exceptions

A coprocessor 2 exception is raised if:

- $cb.tag$ is not set;
- cb is sealed;
- $cb.perms.Permit\_Store$ is not set;
- $addr + size > cb.base + cb.length$;
- $addr < cb.base$.

An address error exception during store (AdES) is raised if:

- $addr$ is not aligned.

Notes

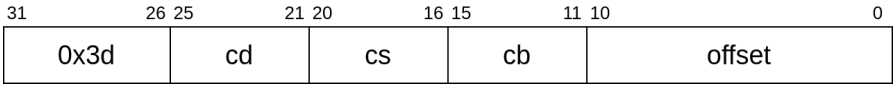
- rt is treated as an unsigned integer.
- $offset$ is treated as a signed integer.
- if $offset = -1$ than the capability written to cd will be cb but with the cursor decremented by the number of bytes written.

UCSC

Format

UCSC cd, cs, offset(cb)

Encoding



Description

This instructions stores capability register *cs* into the memory location specified by $cb.base + cb.offset + rt + capability\_size * offset$ if the capability *cb* has the permission to store capabilities. $capability\_size$ indicates the size of a capability in bytes. For the 256-bit capability format this value will be 32.

When the given offset equals -1, the cursor of the capability *cb* is decremented by the $capability\_size$ and the modified capability with the decremented cursor is written to *cd*. If the offset is not -1, *cb* is copied to *cd*.

Semantics

```
checkCP2usable();
let cs_val = readCapReg(cs);
let cb_val = readCapRegDDC(cb);
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_store) then
  raise_c2_exception(CapEx_PermitStoreViolation, cb)
else if not (cb_val.permit_store_cap) then
  raise_c2_exception(CapEx_PermitStoreCapViolation, cb)
else if not (cb_val.permit_store_local_cap) & (cs_val.tag) & not (cs_val.
    ↪ global) then
  raise_c2_exception(CapEx_PermitStoreLocalCapViolation, cb)
else
{
  let cursor = getCapCursor(cb_val);
  let vAddr = (cursor + cap_size * signed(offset)) % pow2(64);
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + cap_size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if vAddr < getCapBase(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
  else if (vAddr % cap_size) != 0 then
    SignalExceptionBadAddr(AdES, vAddr64)
  else
  {
```

```
    let (pAddr, macr) = TLBTranslateC(vAddr64, StoreData);
    let mtag : bool = match (if cs_val.tag == false then Unrestricted else
        ↪ macr) {
      Unrestricted => cs_val.tag,
      Clear => false,
      Trap => raise_c2_exception(CapEx_TLBNoStoreCap, cs)
    };
    if cb_val.uninit & offset == ones() then {
      let decr = to_bits(64, negate(cap_size));
      let (_, newCap) = incCapOffset(cb_val, decr);
      writeCapReg(cd, newCap);
    } else {
      writeCapReg(cd, cb_val);
    };
    MEMw_tagged(pAddr, cap_size, mtag, capToMemBits(cs_val))
  }
}
```

Exceptions

A coprocessor 2 exception is raised if:

- $cb.tag$ is not set;
- cb is sealed;
- $cb.perms.Permit\_Store$ is not set;
- $cb.perms.Permit\_Store\_Capability$ is not set;
- $cb.perms.Permit\_Store\_Local$ is not set and $cs.tag$ is set and $cs.perms.Global$ is not set;
- $addr + size > cb.base + cb.length$;
- $addr < cb.base$.

A TLB store exception is raised if:

- $cs.tag$ is set and the S bit in the TLB entry for the page containing $addr$ is not set.

An address error exception during store (AdES) is raised if:

- The virtual $addr$ is not $capability\_size$ aligned.

Notes

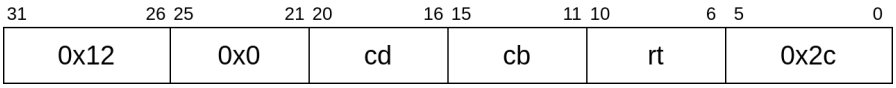
- $offset$ is treated as a signed integer.
- if $offset = -1$ than the capability written to cd will be cb but with the cursor decremented by the $capability\_size$.

CShrink

Format

CShrink cd, cb, rt

Encoding



Description

The capability written to register *cd* will have a different range of authority, specified by:

- $cd.length = cb.offset$;
- $cd.base = rt$.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let base = getCapBase(cb_val);
let newBase = unsigned(rGPR(rt));
let cursor = getCapCursor(cb_val);
let top = getCapTop(cb_val);
if cb_val.sealed then
    raise_c2_exception(CapEx_SealViolation, cb)
else if newBase < base then
    raise_c2_exception(CapEx_LengthViolation, cb)
else if top < cursor then
    raise_c2_exception(CapEx_LengthViolation, cb)
else
{
    let (success, newCap) = shrinkCap(cb_val, to_bits(64, newBase), to_bits(65,
        ↪ cursor));
    if (success) then
        writeCapReg(cd, newCap)
    else
        writeCapReg(cd, unrepCap(newCap))
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is sealed;
- $rt < cb.base$;
- $cb.base + cb.length < cb.base + cb.offset$.

Notes

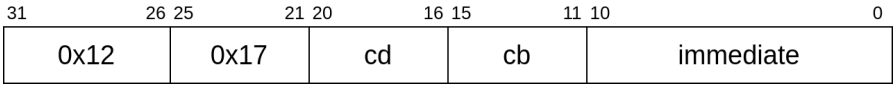
- rt is treated as an unsigned integer;
- $cd.base + cd.offset$ will no longer be in the range of authority of cd.

CShrinkImm

Format

CShrink cd, cb, immediate

Encoding



Description

The capability written to register *cd* will have a different range of authority, specified by:

- $cd.length = cb.offset$;
- $cd.base = cb.base + immediate$.

Semantics

```
checkCP2usable();
let cb_val = readCapReg(cb);
let base = getCapBase(cb_val);
let immU = unsigned(imm);
let cursor = getCapCursor(cb_val);
let top = getCapTop(cb_val);
let newBase = base + immU;
if cb_val.sealed then
    raise_c2_exception(CapEx_SealViolation, cb)
else if newBase < base then
    raise_c2_exception(CapEx_LengthViolation, cb)
else if top < cursor then
    raise_c2_exception(CapEx_LengthViolation, cb)
else
{
    let (success, newCap) = shrinkCap(cb_val, to_bits(64, newBase), to_bits(65,
        ↪ cursor));
    if (success) then
        writeCapReg(cd, newCap)
    else
        writeCapReg(cd, unrepCap(newCap))
}
```

Exceptions

A coprocessor 2 exception is raised if:

- *cb* is sealed;
- $rt < cb.base$;
- $cb.base + cb.length < cb.base + cb.offset$.

Notes

- *immediate* is treated as an unsigned integer;
- $cd.base + cd.offset$ will no longer be in the range of authority of *cd*.

5 Secure Calling Convention

Now that the concrete ISA extension for uninitialized capabilities has been proposed for CHERI-MIPS, we can take a look at the second contribution of this thesis: a secure calling convention for CHERI that uses local and uninitialized capabilities.

The calling convention aims to have the well-bracketed control flow and local state encapsulation properties, just like the calling convention using local capabilities from Section 2.5.

To achieve WBCF, the calling convention prevents adversaries from jumping to return capabilities that the adversary might have stored on the stack, by not allowing adversaries to read contents from the stack unless they have written to that location first (the stored return capability can thus only be overwritten and never read). LSE can be achieved by shrinking the range of authority of the stack capability so that it no longer includes the stack frame of the caller. It is also important that the registers are cleared so that no variables or capabilities concerning the stack frame of the caller can be accessed by the callee.

The secure calling convention is based on the calling convention from Section 2.5 combined with the CHERI calling convention from Section 2.4. The calling convention with local capabilities (Section 2.5) has to be adjusted for the CHERI capability machine (the calling convention was defined for a formal capability machine). The enter capability will be substituted by a sealed capability pair. The capability pair exists of a code capability and data capability. Before invoking a function, the stack capability needs to be made uninitialized to prevent having to clear the unused part of the stack (see Section 2.5.1). The requirement to clear the unused part of the stack before returning to the caller of the function is also no longer required (see Section 2.5.5). The calling convention described in this section is designed for the CHERI capability machine but should apply to capability machines in general that support local capabilities, sealed capabilities and uninitialized capabilities. An important notion for this calling convention is that the stack capability will have the permissions **RWL** and will add the **U** permission on function invocation, resulting in a stack capability with the **URWL** permissions. Figure 5 shows the concept of the stack using an uninitialized stack capability. The red area is not in the range of authority of the stack capability, the green area can be read and written to and the orange area is write-only.

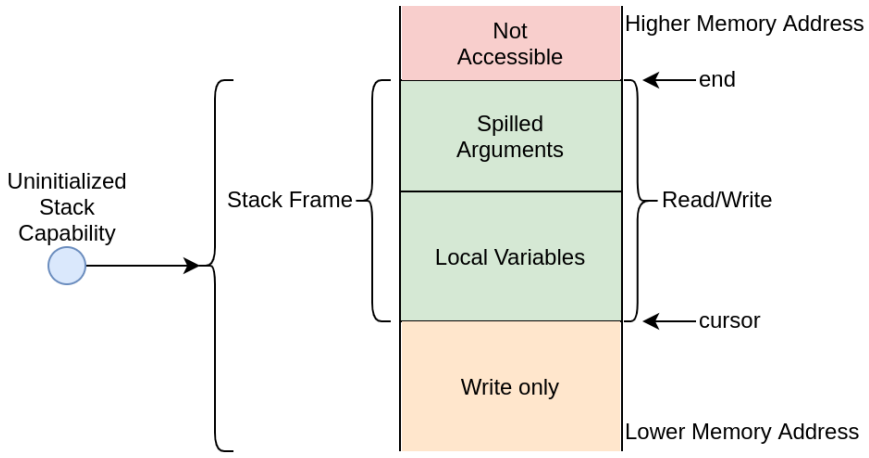


Figure 5: Stack with Uninitialized Capability

Having an uninitialized stack capability prevents adversaries from reading from the stack unless they first overwrite the uninitialized data (this could be garbage but also sensitive data

or capabilities they should not get access to), this scenario is shown in Figure 6.

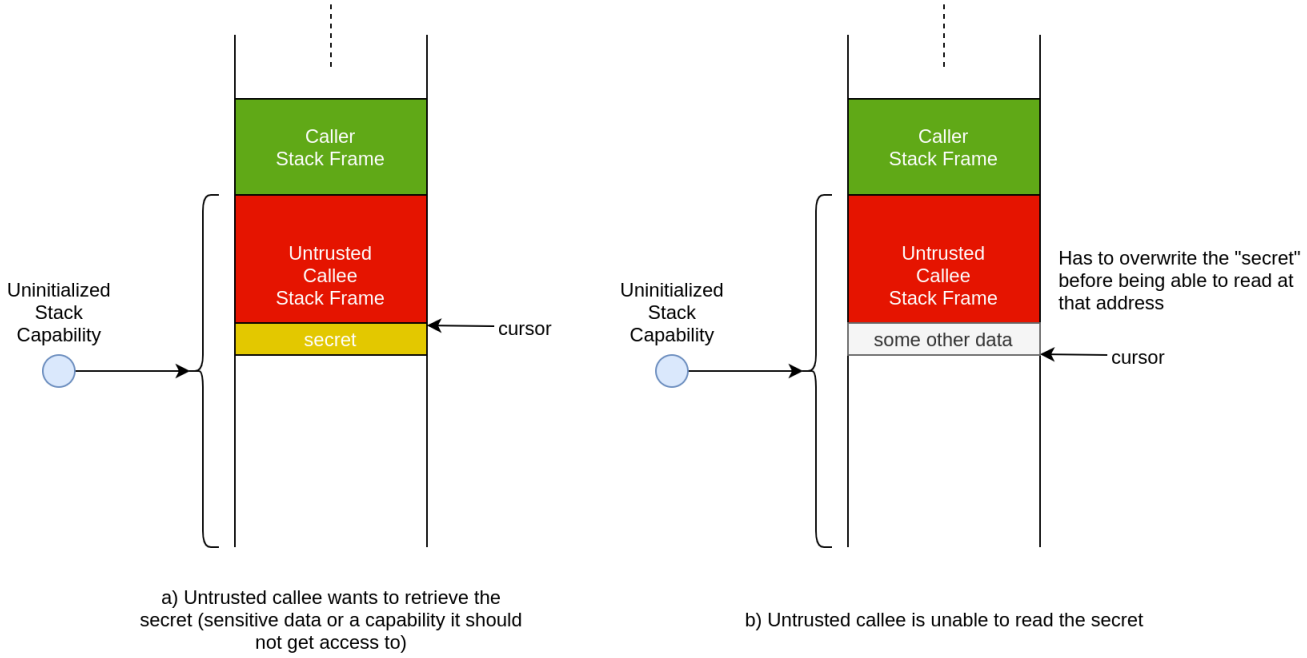


Figure 6: Adversaries cannot read without overwriting first

The calling convention requires sealing a pair of capabilities (the return capability and stack capability) using a unique seal. This pair of capabilities will be passed in registers to the callee, which can only return to its caller by jumping using the given return and stack capability pair. Note that the stack capability used in this jump is the stack capability of the caller, not the callee itself. The unique seal (the callee can never reconstruct the seal) will prohibit adversaries from constructing capabilities with the same seal, in an attempt to supply a different stack capability or return capability. The sealing requirement of the return and stack capability can be fulfilled by using the CCall instruction (described in Section 2.2.1). It is not necessary to be able to unseal the pair, i.e. the return and stack capability, (the ccall instruction will do this if the seals of the sealed capability pair is the same). This means that the seals can be obtained by having a central "sealing capability" from which a seal can be used and that seal can then be made unavailable.

For convenience, this sealing capability will be maintained in the *$DDC* register (default data capability), which is not used in the purecap ABI mode of CHERI (for which this calling convention is designed). The *$DDC* register is a hardware register used in the non-purecap ABIs of CHERI to support legacy loads and stores with pointers instead of capabilities [18, Section 2.3.12]. Using *$DDC*, a hardware register, requires copying it into a general-purpose capability register. This can be done with the *CGetDefault* instruction and setting it with the *CSetDefault* instruction. A general-purpose capability register could be used instead of *$DDC*, as long as *$DDC* gets invalidated (the default data capability can otherwise be used to unseal any sealed capability). A last alternative to the *$DDC* and general-purpose capability register to maintain a "sealing capability" would be to store that sealing capability in memory, so that every function can access it. Obtaining a unique seal is then as trivial as just retrieving the sealing capability to be used in the cseal instructions. Making the just used seal unobtainable

can then be done by increasing the cursor and changing the range of authority of the sealing capability to no longer include the previous cursor (i.e. settings its base address higher). The following sequence of instructions performs these actions:

```
1  # Load seal capability
2  cgetdefault $c13
3
4  # Modify sealed capability
5  cmove $c3, $c13
6  cincoffset $c3, $c3, 1
7  cgetlen $t2, $c3
8  cgetaddr $t3, $c3
9  sub $t2, $t2, $t3 # calculate new length
10 csetbounds $c3, $c3, $t2
11
12 # Store modified seal capability
13 csetdefault $c3
```

On line 2 the sealing capability is loaded into *$c13* (the *cgetdefault* instruction loads the capability in *$ddc* into the specified registers, *c13* in this case). Lines 5-10 modify a copy of the sealed capability (*$c3), so that the seal of /$c13* cannot be reused and is unique. On line 13 the modified sealing capability is stored in the *$ddc* register. The cseal instructions following this sequence of instructions can now use *$c13* to seal capabilities with a unique seal. No other capability will be obtainable that can be used to unseal the sealed capabilities using *$c13* if *$c13* is cleared when it is no longer needed (given that if it is copied to another register(s) that the corresponding register(s) also get cleared). The only remaining problem is that *$c13* might get stored in the heap, making it possible to retrieve it from adversary code. To avoid this, the sealing capability is also made local. Now it can only be stored on the stack and in combination with using uninitialized stack capabilities it is possible to prevent other code from accessing it.

In the following subsections it will be assumed that the stack capability is already a local stack capability and that the sealing capability is local and available in the *$ddc* register.

Note that when a function allocates arrays or structs on the stack, the *CDropUninit* instruction can be used once the array or struct has been completely initialized (the memory needed by the array or struct will be zeroed by the uninitialized stack capability). This makes it easier to work with the array or struct in the rest of the code, for example when using a pointer to loop over an array but instead of looping from the first element towards the last, looping from the last element towards the first (this requires decrementing the offset of the capability, which is not possible for an uninitialized capability).

5.1 Invoking A Function

Before invoking a function, the caller-saved registers should be saved on the stack. Getting the entry point capability for the function to be invoked remains the same as in the original CHERI calling convention. One important caveat is that stack frames should begin at addresses that are multiples of 32, because capabilities can only be stored in memory at addresses that are multiples of 32. If stack frames could begin at any possible address there would be extra overhead every time bytes need to be stored to memory that need to be alignment (for example capabilities need to be stored at addresses that are a multiple of 32), due to adding instructions that would need to check if the alignment rule is being followed and if not write some zeroes first. The stack frame address requirement means that if necessary, some zeroes need to be written on the stack

to make sure that the cursor of the stack capability that will be passed to the callee is a multiple of 32. Therefore the safe assumption can be made that every stack frame will have a size that is a multiple of 32. This requirement also makes it easier to determine in the compiler where to store the local state of a function.

There are a few steps that need to be taken before jumping to the function:

- the current stack capability needs to be sealed;
- the stack capability, to be used by the callee, needs to be shrunk to no longer include the stack frame of the caller (i.e. the unused part of the stack) and needs to be made uninitialized;
- the return capability is constructed as the current program counter capability but incremented so that it returns to the instruction that restores the old stack capability this means passed the register clearing and jumping to the function (the instruction to restore the old stack capability simply moves the contents of *$idc*, the invoked data capability register, into the stack capability register, *$c11*);
- arguments to be passed on the stack can now be pushed onto the stack frame of the callee;
- registers need to be cleared, argument registers that are used should not be cleared, capability registers *$ddc*, *$c1-2*, *$c11-12* should not be cleared (*$ddc* is the capability used for sealing the stack and return capabilities, *$c1-2* will contain the return and stack capability, *$c12* contains the capability to jump to and *$c11* contains the stack capability to be used by the callee);

After performing these steps, the jump to the function can take place. To prevent adversaries from storing either the return or stack capability in global memory, they should both be local capabilities.

The return capability is passed in register *$c1* and the stack capability of the caller in register *$c2*, this choice was made to conform with the CHERI ABI and CheriBSD, which limits the number of registers that may be used with the ccall instruction in order to avoid the need to decode the instruction and to determine the concrete argument registers [18, page 184].

When the callee returns, the stack capability of the caller will be in the *$idc* capability register (which is *$c26*, used as the invoked data capability register) and the first instruction executed by jumping to the return capability is moving the contents of *$idc* into *$c11* to restore the stack capability. The caller-save registers can be restored at this point.

The modified sequence of instruction before jumping to a function then corresponds to:

```
1  cseal $c1, $c11, $c13
2  cshrink $c11, $c11, 0
3  cuninit $c11, $c11
4
5  # store arguments into registers if necessary
6  ...
7  # spill arguments to the stack if necessary
8  ...
9
10 li $t0, 32
11 li $t1, 0xfffffffe
12 cgetpccincoffset $c17, $t0
13 candperm $c17, $c17, $t1
```

```
14 cseal $c1, $c17, $c13
15
16 clearlo 0xffff
17 clearhi 0xffff
18 cclearlo 0b1110011111111000
19 cclearhi 0xffff
20
21 cjr $c12
22 nop
23
24 cmove $c11, $idc
```

The instruction sequence assumes that a unique seal for this function call site is available from the capability in register *$c13*. The first 3 instructions (lines 1-3) seal the stack capability with the unique seal, then shrink the stack capability to only cover the unused part of the stack and make the stack capability now uninitialized. On lines 10-14 the return capability is created and its permissions adjusted so that it is a local capability. The return capability is sealed using the same unique seal as for the stack capability. Note that the program counter is incremented by 32, this number depends on the number of instructions required to skip to get to the cmove instruction (line 24), this will always be 32 (there is one *candperm* instruction, one *cseal* instruction, 4 register clearing instructions, one jump instruction and one no-op instruction, this is a total of 8 instructions and each instruction requires one word, which is 4 bytes, i.e. $8 * 4 = 32).

The register clearing happens on lines 16-19 using the *CClearRegs* [18, page 193] instruction (*clearlo*, *clearhi*, *cclearlo* and *cclearhi* are mnemonics for *cclearregs* that indicate a register set to be cleared). CClearRegs takes as a first argument the register set to be cleared. The second argument is a mask indicating which registers of the register set to clear, when a bit of the mask is set, the corresponding register will be cleared (for example, if bit 0 of the mask is set than the lowest numbered register in the register set will be cleared) Clearlo targets the register set *$0-15*, clearhi register set *$16-31*, cclearlo register set $ddc, *$c1-15* and cclearhi register set *$c16-31*. In the instructions shown, all registers except for *$ddc*, *$c1-2* and *$c11-12* are cleared. When arguments are passed in registers to a function than those registers should not be cleared (i.e. their corresponding bit in the mask should not be set).

The jump to the function to be invoked happens on line 21. Following this jump is a *nop* (no-operation) instruction, which is used for the branch delay slot of the preceding jump instruction. The instruction in the branch delay slot always executes after the branch and is used for better instruction level parallelism with pipelining [10, page 322]. One of the clear registers could possibly be put in this branch delay slot. I have chosen not to do this here as it makes the different steps of the calling convention more clear.

Finally, the instruction on line 24 moves the content of register *$idc* into the stack capability register *$c11*.

5.2 Argument Passing

Argument passing remains the same for this calling convention as in Section 2.4.

5.3 Returning From A Function

Returning from a function requires that the stack frame of the callee (the function from which we want to return) is cleared, thus the local variables and spilled arguments should be cleared.

Unlike the current CHERI calling convention, it is no longer possible to use the *CJR* instruction because the return capability is a sealed capability (passed in *$c1*). Also keep in mind that the previous stack capability is also a sealed capability (passed in *$c2*). The only way to return to the caller is thus by using the *CCall* instruction. As arguments to the *CCall* instruction the return capability and previous stack capability registers are provided. The selector, used to choose how the ccall instruction jumps to the given code and data capability pair, for *CCall* is 1, meaning that the return capability is unsealed and placed in the program counter capability register and the previous stack capability is unsealed and placed in the invoked data capability register (*$idc*).

As mentioned before, the return capability is passed in register *$c1* and the stack capability in register *$c2*.

5.4 Function Prologue

The stack setup happens before the function invocation by the caller, no additional processing is required in the function prologue except for checking that the stack capability has the **URWL** permissions.

This can be done by taking the uninitialized bit and the global bit. It is not necessary to check explicitly for the read and write permissions as these are already needed to make a capability uninitialized. The following assembly code illustrates how to check if the stack capability is uninitialized and local:

```
1 cgetuninit $t0, $c11
2 cgetperm $t1, $c11
3 not $t1, $t1
4 andi $t1, $t1, 1
5 and $t1, $t1, $t0
6 teq $t1, $zero
```

On the first two lines the uninitialized bit is written to register *$t0*. The permissions of the capability are stored in *$t1*. The *not* instructions is required because, as mentioned earlier, a capability is local if its *global* bit is not set. By using the *not* instruction the global bit will now be set and the *andi* instruction will keep it set in *$t1* (if the stack capability was local, otherwise *$t1* will now be all zeros). The *and* instruction on line 5 is used to combine the uninitialized bit and local bit. If the capability is uninitialized and local then *$t1* will not be all zeros. The last instruction, *teq*, will trap if *$t1* is all zeros, i.e. the stack capability was not local and uninitialized.

5.5 Function Epilogue

The instruction sequence for returning from a function in the secure calling convention becomes:

```
1 # Clear local variables and spilled arguments
2 ucsw $c11, $zero, 1($c11)
3 ucsw $c11, $zero, 0($c11)
4
5 # Clear non-return registers
6 clearlo 0b1111111111111011
7 clearhi 0xffff
8 cclearlo 0b1111111111111001
9 cclearhi 0xffff
```

```
10
11 ccall $c1, $c2, 1
```

The first lines (1-3) indicate that the local variables and spilled arguments should be cleared, this happens by writing the value of the zero register (*$0* which is the same as *$zero*) over the memory containing the local variables and spilled arguments until they are completely cleared. In the example instructions above there were 2 local variables in the stack frame and these have been overwritten with zeroes.

Lines 6-9 clear the non-return registers. The only registers not cleared are *$c1* and *$c2* (as these will be used for the *CCall* instruction) and optionally the registers containing the return value, which can be *$c3* if a capability is returned or *$2-3* for non-capability return values. In the instruction sequence shown above, the assumption is made that the return value is stored in *$2*, i.e. its bit in the mask to *clearlo* is not set.

The *CCall* instruction follows on line 11 and performs the actions discussed earlier. Unlike other jump instructions in MIPS, it does not have a branch delay slot.

6 Assembler

This section will describe the modifications made to the CHERI-MIPS assembler in LLVM. Every file mentioned in this section assumes that the current working directory is *llvm-project/llvm/lib/Target/Mips* (where llvm-project is the root directory of the LLVM repository), unless specified otherwise. The assembler will be used later on in the evaluation section.

In Section 4.1.3 the new instructions were listed and explained. These were then implemented in the CHERI-MIPS simulator. These instructions still need to be added to the assembler so that they can be correctly encoded into a binary executable file.

The encoding of the new instructions is specified in *MipsInstrCheri.td*. The following code fragment shows the encoding of the *CGetUninit*, *CUninit* and *CShrink* instructions:

```
1  def CGetUninit : CheriFmtCGet<0x15, "uninit">;
2
3  def CUninit : CheriFmt2Op<0x1b, (outs CheriOpnd:$r1), (ins CheriOpnd:$r2),
4      "cuninit\t$r1, $r2",
5      [(set CheriOpnd:$r1, (int_cheri_cap_uninit CheriOpnd:$r2))]>
6
7  def CDropUninit : CheriFmt2Op<0x1c, (outs CheriOpnd:$r1), (ins CheriOpnd:$r2),
8      "cdropuninit\t$r1, $r2",
9      [(set CheriOpnd:$r1, (int_cheri_cap_drop_uninit CheriOpnd:$r2))]>
10
11 def CShrink : CheriFmt3Op<0x2c, (outs CheriOpnd:$r1), (ins CheriOpnd:$r2,
       ↪ GPR64Opnd:$r3),
12     "cshrink\t$r1, $r2, $r3",
13     [(set CheriOpnd:$r1, (int_cheri_cap_shrink CheriOpnd:$r2, GPR64Opnd:$r3))]>;
```

The language used is *TableGen*, which can generate *C++* code for the definitions and records that are defined in a tablegen file. The above code fragment has 3 definitions. The first definition, *CGetUninit* is an instance of the record *CheriFmtCGet*, which only requires the value of the *func* field of the instruction (0x15 in this case) and the name of the instruction (without the *cget* prefix). This definition will expand to a definition similar to that of *CUninit*, with as intrinsic $int_{cheri cap uninit get}$. This intrinsic specifies that the instruction takes one general purpose out register and one capability in register, it also mentions that the instruction does not access memory. The *CUninit* definition specifies that it is an instance for a *CheriFmt2Op* (cheri format for a 2 operands instruction) record, with as value for the func field 0x1b. It has one out register, *$r1* and one in register *$r2*. The assembly format is specified next, followed by some intrinsics about the registers. The first intrinsic specifies that *$r1* will be set (i.e. written to) and the second intrinsic, $int_{cheri cap uninit}$ says that the instruction has one out capability register and one in capability register. The intrinsic also specifies that it does not interact with memory (it only uses registers). *CDropUninit* is similar to *CUninit*, with the only difference being the assembly format and the memory intrinsic used (but the $int_{cheri cap drop uninit}$ is equivalent to the $int_{cheri cap uninit}$ intrinsic). The *CShrink* definition specifies it is of the *CheriFmt3Op* format, the cheri format for 3 operands instructions, with a single out register and 2 in registers. The intrinsic $int_{cheri cap shrink}$ specifies it has one out capability register, one in capability register and one in general purpose register. The intrinsic also specifies it does not access memory.

The intrinsics, $int_{cheri cap uninit get}$, $int_{cheri cap uninit}$ and $int_{cheri cap shrink}$ are given for completeness (in *llvm-project/llvm/include/llvm/IR/IntrinsicsCheriCap.td*):

```
def int_cheri_cap_shrink:
    Intrinsic<[llvm_fatptr_ty],
              [llvm_fatptr_ty, llvm_anyint_ty],
              [IntrNoMem, IntrWillReturn]>;

def int_cheri_cap_uninit_get :
    Intrinsic<[llvm_anyint_ty],
              [llvm_fatptr_ty],
              [IntrNoMem, IntrWillReturn]>;

def int_cheri_cap_uninit :
    Intrinsic<[llvm_fatptr_ty],
              [llvm_fatptr_ty],
              [IntrNoMem, IntrWillReturn]>;
```

Before diving into the definitions of the uninitialized store instructions, I will go over the format they use, *CheriFmtUCSX*, defined in *MipsInstrFormatsCheri.td*:

```
class CheriFmtUCSX<bits<2> t, bits<1> e, dag outs, dag ins, string asmstr,
                   list<dag> pattern> :
   MipsInst<outs, ins, asmstr, pattern, NoItinerary, FrmOther>, Sched<[]>
{
    bits<5> cd;
    bits<5> rs;
    bits<5> cb;
    bits<8> offset;

    let Opcode = 0x3b;
    let Inst{25-21} = cd;
    let Inst{20-16} = rs;
    let Inst{15-11} = cb;
    let Inst{10-3} = offset;
    let Inst{2} = e;
    let Inst{1-0} = t;
    let hasSideEffects = 1;
}
```

This class corresponds to the encoding of the uninitialized store instructions (Figure 7).

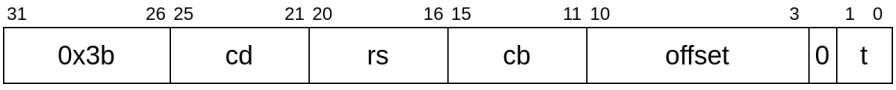


Figure 7: UCSX Encoding

The bit specified for the *e* field will always be 0, but is passed as parameter in case the extra bit would be needed for something else in the future.

The uninitialized store instructions, added in *MipsInstrCheri.td* are defined as:

```
multiclass UStoreViaCapScaled<bits<2> t, bit e, string instr_asm,
    ↪ RegisterOperand RC, PatFrag storeType, Operand simm, PatFrag immfrag> {
```

```
2   def #NAME# : CheriFmtUCSX<t, e,
3                (outs CheriOpnd:$cd),
4                (ins RC:$rs, simm:$offset, CheriOpnd0IsDDC:$cb),
5                !strconcat(instr_asm, "\t$cd, ${rs}, ${offset}(${cb})"),
6                [(set CheriOpnd:$cd, (storeType (CapRegType (ptradd
                     ↪ CheriOpnd0IsDDC:$cb, (i64 immfrag:$offset))), RC:$rs))]>;
7   }
8
9   multiclass UStoreViaCap<bits<2> t, bit e, string instr_asm, RegisterOperand RC,
        ↪ PatFrag storeType> :
10    UStoreViaCapScaled<t, e, instr_asm, RC, storeType, simm8, immSExt8>;
11
12  defm UCAPSTORE8 : UStoreViaCap<0, 0, "ucsb", GPR64Opnd, truncstorei8>;
13  defm UCAPSTORE16 : UStoreViaCap<1, 0, "ucsh", GPR64Opnd, truncstorei16>;
14  defm UCAPSTORE32 : UStoreViaCap<2, 0, "ucsw", GPR32Opnd, store>;
15  let isCodeGenOnly=1 in {
16    defm UCAPSTORE832 : UStoreViaCap<0, 0, "ucsb", GPR32Opnd, truncstorei8>;
17    defm UCAPSTORE1632 : UStoreViaCap<1, 0, "ucsh", GPR32Opnd, truncstorei16>;
18    defm UCAPSTORE3264 : UStoreViaCap<2, 0, "ucsw", GPR64Opnd, truncstorei32>;
19  }
20  defm UCAPSTORE64 : UStoreViaCap<3, 0, "ucsd", GPR64Opnd, store>;
```

Two multiclasses are defined, *UStoreViaCapScaled* and *UStoreViaCap* that show the format of the instruction (*CheriFmtUCSX*). This format has one out capability register, one input general purpose register, a signed immediate (the offset) and one more capability register as input. The intrinsics say that register *$cd* will be written to and that the value of *$rs* will be stored in the memory location calculated from *$cb* and the offset. Note that in register *$cb* will be the *$DDC* register if the register is *$c0* (the null register), this is for the legacy MIPS load and store instructions. The difference between *UStoreViaCap* and *UStoreViaCapScaled* is that the latter makes it possible to specify conditions the offset needs to adhere to. In *UStoreViaCap* this is the *simm8* and *immSExt8* values. These values just mean that the offset is a sign extended immediate of 8 bits. Starting on line 12, you see the definitions for the different store instructions (defm is used for definitions of a multiclass). You will notice that there are 2 versions of the *ucsb*, *ucsh* and *ucsw* instructions. This is to handle the case were the MIPS processor has 32-bits general purpose registers.

7 Evaluation

Two kinds of evaluation were performed, the first one being the unit testing of instructions, for the modified and new instructions. The second evaluation is concerned with the new calling convention and how it compares to the calling convention currently in use on CHERI-MIPS.

For both evaluations the same testing infrastructure is used. The tests are added to a fork of the cheritest[3] repository. The cheritest repository[4], developed by the researchers behind CHERI, makes it easy to run tests on the CHERI-MIPS simulator. Tests are defined by an assembly file and python file (both must have the same name, only the file extension should differ). The python file is used to make the test assertions based on the log file produced by running the machine code (compiled from the assembly file) on the CHERI-MIPS simulator. The log file contains the contents of the registers (both the general-purpose registers and the capability registers), how many instructions were executed, the time elapsed for running the instructions (in nanoseconds), how many instructions were executed per second and some other information that is of less relevance for the evaluation.

The following sections will mention where the tests can be found, this is for the fork of the cheritest repository[3]. The tests are run on the C simulator of CHERI-MIPS, as generated by Sail.

7.1 Unit Testing Instructions

The unit tests for the modified and new instructions can be found in `tests/uninitialized_capabilities`. These tests are focused on the behavior of the instructions and they all pass. I will not go over each test individually, this would become rather tedious to read, a small description of each test file can be found in Table 1.

Test File	Description
`test_cshrink`	Test that the bounds of a capability are correctly updated
`test_cdropuninit`	Test that only an uninitialized capability with cursor=base can drop the uninit permission
`test_csealuninit`	Test that a sealed capability can't be mutated by trying to make it uninit
`test_uninit`	Test that the cuninit instruction makes a capability uninitialized
`test_uninit_cap_csetaddr`	Test that the address of an uninitialized capability cannot be lowered
`test_uninit_cap_offset`	Test that offsets are updated correctly and that offset cannot be lowered
`test_uninit_cap_ucstore`	Test that the UCS[BHWD] works as expected
`test_uninit_cap_ucstorecap`	Similar to `test_uninit_cap_ucstore`

Table 1: Unit tests with small description

I will discuss one test in more detail, `test_cshrink`, to give the reader of the thesis a better understanding of how these are written. This will make it easier to inspect the test files yourself and will come in handy for the second evaluation.

[3] https://github.com/capt-hb/cheritest
[4] Original cheritest repository: https://github.com/CTSRD-CHERI/cheritest

The `test_cshrink` test exists of two files, the assembly file containing the instructions to be run on the machine (`test_cshrink.s`) and a python file (`test_cshrink.py`)to make assertions on the log file produced by the simulator.

The assembly file exists of two parts, one part is the test instructions to be executed, wrapped in the `BEGIN_TEST` and `END_TEST` macros and the second part is data part, which will be used to create a capability for and write to/read from. The data part defines a few words and is further not important to understand the test. I will focus on the first part, the test instructions:

```
1  cgetdefault $c1
2  dla $t0, data
3  csetoffset $c1, $c1, $t0
4  csetboundsimm $c1, $c1, 10
5  cincoffsetimm $c1, $c1, 8
6  cgetbase $t0, $c1
7
8  cshrink $c2, $c1, 0 # length of $c2 = offset = 8
9  dli $s0, 10
10 ucsb $c2, $s0, -1($c2)
11 cgetlen $a0, $c1
12 cgetlen $a1, $c2
13 cgetbase $t3, $c2
14
15 csetoffset $c1, $c1, $a0
16 cincoffset $c1, $c1, 1
17 cshrink $c3, $c1, 0 # error: shrinking with offset out of bounds shouldn't work
18
19 cincoffset $c4, $c1, -10
20 cshrink $c4, $c4, $t0
21 cgetbase $t1, $c4
22 cgetlen $a2, $c4
23
24 cincoffset $c5, $c1, -1 # cursor will now be on end
25 cshrink $c5, $c5, 0
26 cgetlen $a3, $c5
27 cshrink $c6, $c5, 1
28 cgetbase $t2, $c6
29 ucsb $c5, $s0, -1($c5)
```

Lines 1-6 setup the capability and store the base of *$c1* into *$t0*. Next, a cshrink instruction is issued to shrink the capability *$c1* to a capability with the same base but with as end the current cursor of *$c1*. A store instruction follows, to make sure no error is thrown after shrinking a capability. Then some values of *$c1* and *$c2* are stored in registers *$a0*, *$a1* and *$t3*. On lines 15-17, the cursor of the capability is set to the end of the capability and right after that is incremented by one. This means that the cursor is now out of bounds ($cursor > end$) and shrinking the capability should raise an error. The remaining instructions are similar to those discussed earlier.

It is now clear what the assembly test instructions do. What remains to be explained is the test cases written in the python file:

```
1  from beritest_tools import BaseBERITestCase
```

```
2
3  class test_cshrink(BaseBERITestCase):
4     EXPECTED_EXCEPTIONS = 1
5
6     def test_cshrink_lowers_end(self):
7        '''Test that lowering the end of a capability works'''
8        assert self.MIPS.a0 == 10
9        assert self.MIPS.a1 == 8
10       assert self.MIPS.a1 < self.MIPS.a0
11       assert self.MIPS.a2 == 1
12
13    def test_cshrink_with_cursor_at_end_does_nothing(self):
14       '''Test that using cshrink with a capability for which cursor=end does not
             ↪ change end'''
15       assert self.MIPS.a0 == self.MIPS.a3
16
17    def test_cshrink_increases_base(self):
18       assert self.MIPS.t0 == self.MIPS.t1
19       assert self.MIPS.t0 == self.MIPS.t3
20       assert self.MIPS.t0 + 1 == self.MIPS.t2
```

The test class needs to be the same name as the file (`test_cshrink`) and subclasses `BaseBERITestCase` (which gives it easy access to the contents of the log file for this test). The `EXPECTED_EXCEPTIONS` is declared to be one (the error that is expected as explained in the assembly file discussion). Three methods are defined, the first methods, `test_cshrink_lowers_end` tests that the cshrink instruction correctly lowers the end of a capability. Here you can see why subclassing `BaseBERITestCase` is useful, the registers can be addressed as `self.MIPS.a0`. The second method, `test_cshrink_with_cursor_at_end_does_nothing` tests that shrinking a capability (in which we don't change the base of the capability) with its cursor already at the end does not change the end of the capability. The last method, `test_cshrink_increases_base` is similar to `test_cshrink_lowers_end`.

7.2 Calling Convention

To evaluate the calling convention, pairs of tests were used. The pair consists of a test for the original calling convention and a test for the new calling convention. These pairs were created for a few example programs. In this section I will go over the different programs used to test the new calling convention and what the results are. The results entail the semantics preservation of the calling convention (using the original and new calling convention produces the same output), how many instructions each assembly file required (remember that a test consists of a python file and an assembly file, and because I use a pair of tests there is an assembly file for each calling convention) and the execution time of the assembly code on the C simulator of CHERI-MIPS. As in the previous section, the complete test files can be found in the cheritest[3] repository, but this time in the *tests/purecap* directory. These tests were written in the purecap directory so that they are executed using the pure capability calling convention. The assembly files are based on the output by running the following command:

```
clang -S <source_file> -mcpu=beri -mabi=purecap -cheri-linker -cheri=256 \
     -target cheri-unknown-freebsd -Wall -fomit-frame-pointer -O0 <output_file>
```

The output file generated by this command is not sufficient to be used as a purecap test, the `main` routine needs to be renamed to `test`. The adjusted assembly file can then be used for the *original* test of the pair. A manual modification of this files was performed to have it use the secure calling convention as described in Section 5.

First, some general results are given concerning the number of instruction of the prologue and epilogue of functions. Then the evaluation continues by going over the considered programs and giving the results of the evaluation of that program. The performance of the assembly code is measured in nanoseconds (the Y-axis on the boxplots) and is repeated 20 times. When the performance is shown in tables or text it will be shown in microseconds for readability.

Each test discussed in the upcoming sections consists of four files, two files for the original calling convention and two for the secure (also referred to as *uninit*) calling convention. The two files per calling convention are the assembly file (*.s* file extension) and the python file (*.py* file extension). The tests can be found in the purecap directory with their full name shown in Table 2, note that `<calling-convention>` is `original` or `uninit`.

Test	Test Files
Simple Function Call	`test_purecap_<calling-convention>_cc_simple_call`
Stack Growth	`test_purecap_<calling-convention>_cc_stack_growth`
Stack Growth -O1	`test_purecap_<calling-convention>_cc_stack_growth_O1`
Sum Factorials	`test_purecap_<calling-convention>_cc_slow_factorial`
Sums -O1	`test_purecap_<calling-convention>_cc_sums_O1`

Table 2: Test files for each test

One more note, the number of instructions is always measured without the comments in the assembly file and the assembler directives are also not included in the count. The instruction count is for the actual number of instructions.

7.2.1 Simple Function Call

The first program contains one single function invocation:

```
1 int doSomething(int a) {
2     return a;
3 }
4
5 int main(void) {
6     int value = doSomething(100);
7     return value;
8 }
```

The function `doSomething` returns the argument it was passed. The `main` function will return the value from the `doSomething` call (which will be 100). The tests for this example are `test_purecap_original_cc_simple_call` and `test_purecap_uninit_cc_simple_call`. In the python test files an assertion is made that the result of the program is 100. Both tests pass, so the calling convention does not alter the semantics of the program. Table 3 shows some interesting statistics about the number of instructions of each calling convention and the performance (the median in microseconds).

The performance is visualized using a box plot in Figure 8.

Calling Convention	Instructions	Performance
Original	39	942.2315
Secure	66	982.0340

Table 3: Results for the simple function call

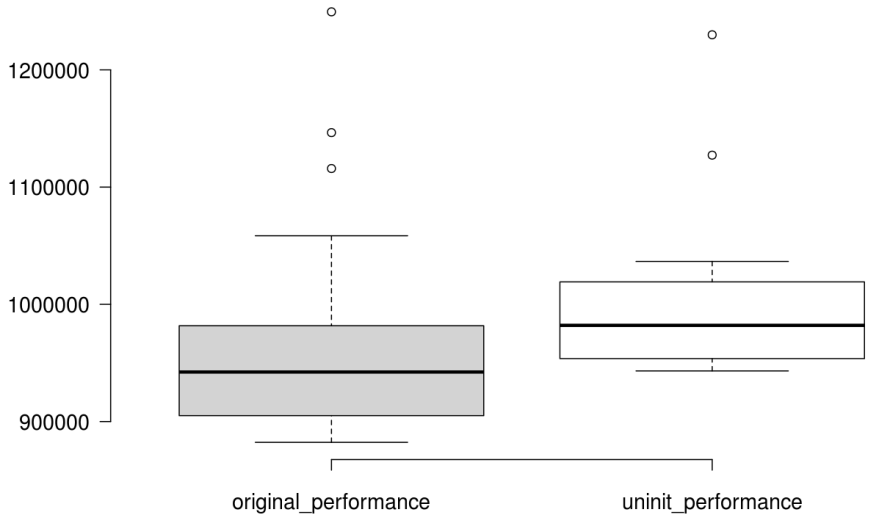


Figure 8: Boxplot performance of the simple call program

From this boxplot and the table above we can conclude that the new calling convention is slower than the original calling convention, which was expected, but not much slower, the overhead is acceptable.

7.2.2 Stack Growth

This program has a function that calls another function, requires spilling arguments to the stack and creates capabilities for variables.

```
1  int g(int *a, int *b) {
2      return (*a) + (*b);
3  }
4
5  int f(int a) {
6      int x = 10;
7      return g(&a, &x);
8  }
9
10 int tmp(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j) {
11     return a + b + c + d + e + f + g + h + i + j;
12 }
13
14 int cap_tmp(int *a, int *b, int *c, int *d, int *e, int *f, int *g, int *h, int
       ↪  *i, int *j) {
15     return (*a) + (*b) + (*c) + (*d) + (*e) + (*f) + (*g) + (*h) + (*i) + (*j);
16 }
17
18 int mixed_tmp(int a, int *b, int c, int *d, int e, int *f, int g, int *h, int i
       ↪ , int *j, int k, int *l) {
19     return a + (*b) + c + (*d) + e + (*f) + g + (*h) + i + (*j);
20 }
21
22 int main(void) {
23     int a = 1;
24     int b = 2;
25     int c = 3;
26     int d = 4;
27     int e = 5;
28     int x = 6;
29     int g = 7;
30     int h = 8;
31     int i = 9;
32     int j = 10;
33     tmp(a, b, c, d, e, x, g, h, i, j);
34     cap_tmp(&a, &b, &c, &d, &e, &x, &g, &h, &i, &j);
35     mixed_tmp(a, &b, c, &d, e, &x, g, &h, i, &j, i, &j);
36     return f(10);
37 }
```

The `tmp` and `cap_tmp` functions are interesting because they require argument spilling to the

stack (remember that only 8 integers can be passed in general-purpose registers and 8 capabilities in capability registers). `mixed_tmp` does not require any argument spilling, it takes 12 parameters but half of them are integers and half capabilities, so these can be passed in registers (6 parameters using general-purpose registers and 6 using capability registers). The `f` function calls `g` with capabilities for its parameter `a` and a local variable `x`.

To test the semantics of this program, not only the value of the main function is considered, but the return values from the function calls to `tmp`, `cap_tmp` and `mixed_tmp` are also tested to make sure their result remains the same in the original calling convention and the new calling convention. The tests pass for both calling conventions, thus the semantics of the program are preserved.

Table 4 shows some interesting statistics about the number of instructions of each calling convention and the performance (the median in microseconds).

Calling Convention	Instructions	Performance
Original	393	1752.0225
Secure	647	2347.0310

Table 4: Results for the stack growth program

The performance is visualized using a box plot in Figure 9.

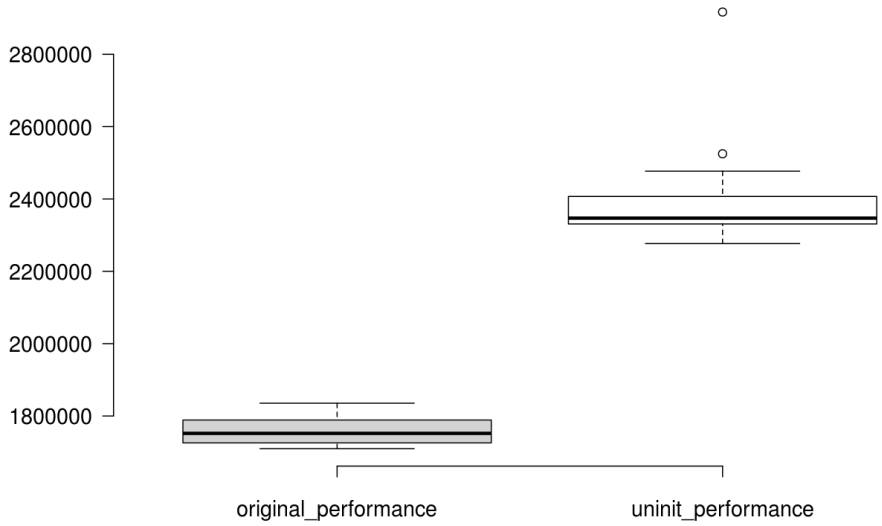


Figure 9: Boxplot performance of the stack growth program

From the boxplot and Table 4 we learn that the program is slower when more function invocations take place. This makes sense because the calling convention poses an overhead for function calls, instruction sequences that do not perform a function invocation or don't return from a function, are only altered for storing contents on the stack, which should have similar performance to the existing store instructions (CS[BHWD], CSC).

7.2.3 Stack Growth -O1

All the other programs used in the evaluation were compiled with optimization level 0, but the stack growth example has also been compiled with optimization level 1. The assembly generated

by the compiler is used for the original test case and has also been modified to use the new calling convention. With this optimization level, the calls to the functions of which the return value is not used are omitted. Stack frames are also used better and are as small as possible. Redundant loads and stores have also been removed now. Table 5 summarizes the results of evaluating the original calling convention and the secure calling convention for this program.

Calling Convention	Instructions	Performance
Original	88	936.3010
Secure	196	1097.5060

Table 5: Results for the stack growth program compiled with -O1

The performance is visualized using a box plot in Figure 10.

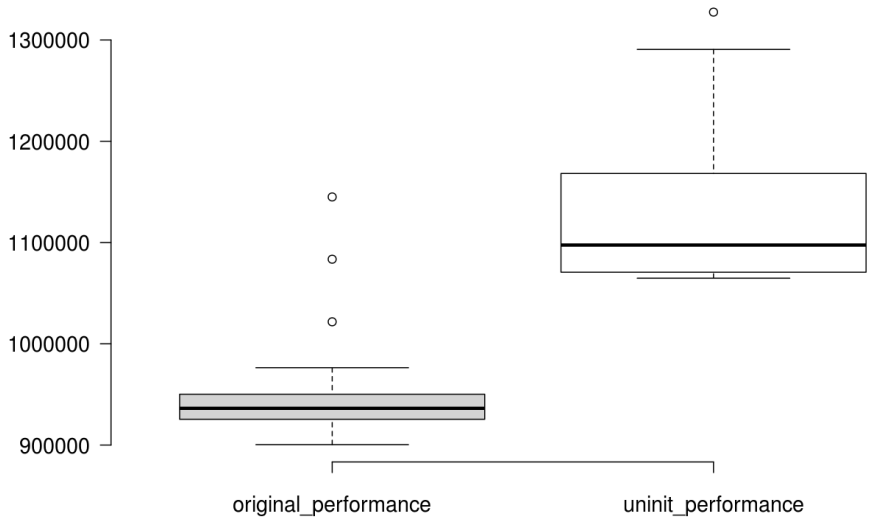


Figure 10: Boxplot performance of the stack growth program compiled with -O1

Even though the number of instructions doubled, the performance isn't that much affected. This is because some of the stack clearing instructions have been removed due to smaller stack frames of individual functions (smaller stack frames require less clearing before returning to the caller of the function).

7.2.4 Sum Factorials

The program for this section calculates the sum of the first four factorials, i.e. $0! + 1! + 2! + 3!$. It uses an array to hold the factorials and after calculating the factorials of each number it adds them together.

```
1 int product(int a, int b) {
2     return a * b;
3 }
4
5 int factorial(int n) {
```

```
6       int total = 1;
7
8       for (int i = n; i > 1; i--) {
9           total = product(total, i);
10      }
11
12      return total;
13  }
14
15  int sum(int nums[], int length) {
16      int sum = 0;
17
18      for (int i = 0; i < length; i++) {
19          sum += nums[i];
20      }
21
22      return sum;
23  }
24
25  int sumFactorials() {
26      int length = 4;
27      int fact[4];
28      fact[0] = 1;
29
30      for (int i = 1; i < length; i++) {
31          fact[i] = factorial(i);
32      }
33
34      return sum(fact, length);
35  }
36
37  int main(void) {
38      return sumFactorials();
39  }
```

The `product` function is defined to have some more function calls. The code should be easy to follow. Just as in the other tests, for both calling convention the return value of the main function is tested to be 10. This semantics preserving tests passes for both calling conventions. Table 6 shows the number of instructions in the assembly code and the execution time (performance column, in microseconds) for each calling convention.

Calling Convention	Instructions	Performance
Original	205	1977.5115
Secure	412	2802.7360

Table 6: Results for the sum of factorials program

The performance is visualized using a box plot in Figure 11.

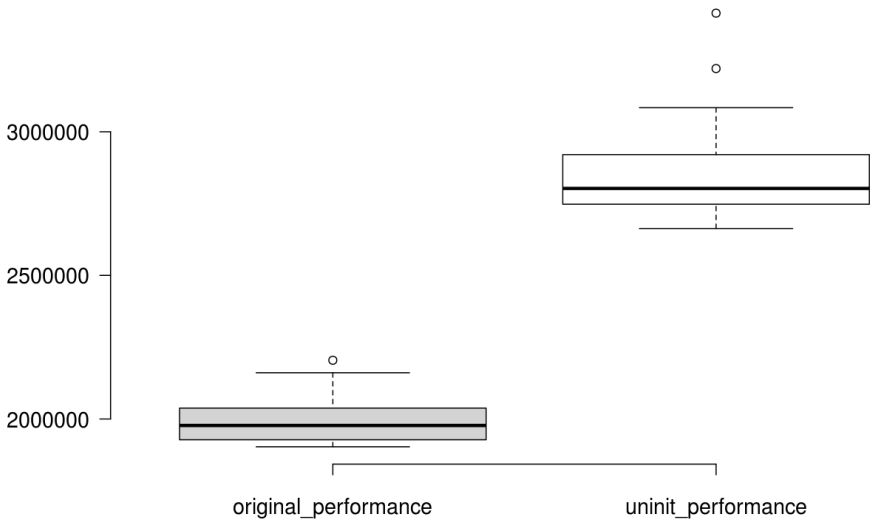


Figure 11: Boxplot performance of the sum of factorials program

The results from this evaluation are similar to the previous ones. There number of instructions has doubled and the overhead introduced by the secure calling convention is visible in the execution time.

7.2.5 Sums -O1

One more program is considered, also compiled with optimization level 1.

```
1  #define LENGTH 10
2
3  void integers(int arr[], int length, int start) {
4      for (int i = 0; i < length; i++) {
5          arr[i] = start + i;
6      }
7  }
8
9  int sum(int *arr, int length) {
10     int total = 0;
11
12     for (int *p = arr; p < arr + length; p++) {
13         total += *p;
14     }
15
16     return total;
17 }
18
19 int backwards_sum(int *arr, int length) {
20     int total = 0;
21
```

```
22      for (int *p = arr + length - 1; p >= arr; p--) {
23          total += *p;
24      }
25
26      return total;
27  }
28
29  int subtract_sums() {
30      int arr[LENGTH];
31      integers(arr, LENGTH, 1);
32      return sum(arr, LENGTH) - backwards_sum(arr, LENGTH);
33  }
34
35  int main() {
36      return subtract_sums();
37  }
```

This program will compute the sum of the elements in an array, once starting from the first element, in the function `sum`, and once starting from the end of the array, in the function `backwards_sum`. The return value of the `main` function is the result of calling `subtract_sums`, which creates a local array on the stack, populates it with the integers starting from 1 and subtracts the sums of the two sum functions. Having a loop start from the end of the array and decrement the pointer in the array is quite interesting, it requires that the capability decrements the offset with the size of the elements stored in the array. Uninitialized capabilities however, do not allow decrementing the offset, this can only be achieved by storing on the address immediately below the cursor. This is where the *CDropUninit* instruction comes in handy, after the array is allocated on the stack, the capability for that array will have its cursor equal to its base, i.e. it has been initialized (with zeroes). This means that the *CDropUninit* instruction can be used to make the capability for the array drop the uninitialized permission and becoming a read, write-local local capability. Decrementing the offset is possible again and this program will run as intended. As in the other examples, tests are setup so that the result from the program using the original and secure calling convention is both zero (the result of `subtract_sums` should clearly be zero). The test passes for both calling conventions. Table 7 summarizes the results of evaluating the original calling convention and the secure calling convention for the sums program.

Calling Convention	Instructions	Performance
Original	87	1292.1815
Secure	250	1589.1595

Table 7: Results for the sums program compiled with -O1

The performance is visualized using a box plot in Figure 10.

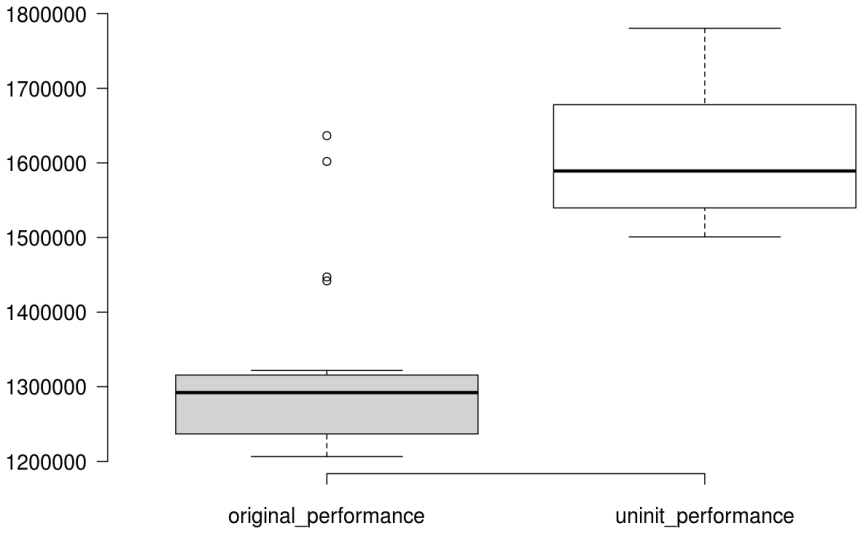


Figure 12: Boxplot performance of the sums program compiled with -O1

The number of instructions has more than doubled for this program but the performance overhead is acceptable, this is because compiling with some optimization can reduce the size of stack frames, which in turn requires less stack clearing when a function returns to its caller.

7.2.6 Conclusion

The performance of the programs discussed is slower using the secure calling convention. This is an expected observation, the secure calling convention requires clearing registers and that a called function clears its own stack frame before returning to the caller. The number of instructions doubles when using the secure calling convention, this is due to

- getting a unique seal for each function invocation;
- setting up the stack and return capability for the function to call;
- clearing registers before jumping to the function;
- clearing the stack frame before returning to the caller;
- clearing registers before returning to the caller.

The overhead is still an improvement when compared to the calling convention using local capabilities as described in Section 2.5, which required clearing the unused part of the stack (the calling convention with uninitialized capabilities only requires that a function clears its own used stack frame). Also keep in mind that most of the programs were compiled with no optimization, in the two programs that were compiled with optimization level one, the overhead is less because the stack frames were smaller, resulting in less stack clearing before a function returns to its caller.

Furthermore, clearing the registers using the *CClearRegs* instruction might be faster in future updates to CHERI-MIPS. Instead of actually writing zeros to general-purpose registers or the

null capability to capability registers, a bit per register could be used to indicate if it is valid or not. This bit would be cleared when clearing a register and set on subsequent writes [18, page 194].

8 CLang Exploration

Due to the time restrictions of the master thesis I was unable to modify the CLang/LLVM compiler. As an alternative, this section will describe the foreseen modifications that will need to be made in the compiler. This section should thus be seen as an exploration of the modifications needed to implement the calling convention in the CLang/LLVM compiler. The files mentioned in this section are relative to the MIPS backend in the LLVM project, *llvm-project/llvm/lib/Target/Mips*. Most of the subsections discussed here correspond to those defined in Section 5. Note that some sequences of instructions could potentially be wrapped in pseudo instructions to make it more clear what the intention is. For example, getting the unique seal from the sealing capability requires quite a few instructions and it might be beneficial to wrap this in a pseudo instruction which takes a single argument in which the capability to use as a seal will be stored.

8.1 Calling Convention

First, a calling convention similar to the CHERI purecap calling convention needs to be defined. This can be done in *MipsCallingConv.td*. In the code that follows an enum value will be available for the new calling convention, for example if we defined the calling convention with the name `CC_CHERI_PURECAP_UNINIT`, we will have the enum `CallingConv::CHERI_PURECAP_UNINIT`. Using this enum value we can adjust the code to do things differently if the used calling convention corresponds with `CallingConv::CHERI_PURECAP_UNINIT`. Alternatively, we could view the current calling convention as the new calling convention and alter the code with the modifications described below always taking place (i.e. the calling convention is always used).

8.2 Store Instructions

Stores using capabilities currently happen using the *CSX* store instructions (those already provided by CHERI-MIPS before uninitialized capabilities were added). These instructions will now have to be the new store instructions (*UCSX*). This change will also require that stores be done differently. The *CSX* store instructions have no out register, but instead have another general-purpose register that is added to the offset of the capability. A concrete example involves writing to an address in an array:

```
csw $2, $1, 0($c2)
```

Assume that *$2* is the register containing the value we want to write, *$1* is the address within the array (i.e. the $index * size_of_elements$) and *$c2* is the capability for the array, with its cursor at the base. This realistic example cannot be done directly using the uninitialized store instructions. It requires one additional instruction:

```
cincoffset $c2, $c2, $1
ucsw $c2, $2, 0($c2)
```

The added instruction, `cincoffset $c2, $c2, $1`, puts the cursor at the address in the array to which we want to write the contents of *$2*.

This example shows that instructions using the *CSX* store instructions can always be transformed into 2 instructions using the uninitialized store instructions. For stack related loads and stores, the instructions are of the form `csw $2, $zero, 16($c2)`, where *$zero* is the register containing all zeroes and *$c2* the stack capability. These instructions can be directly translated to use the uninitialized store instruction: `ucsw $c2, $2, 16($c2)`.

8.3 Invoking A Function

The modifications for invoking a function will take place in *MipsISelLowering.cpp*, in the `LowerCall` function. Nodes will need to be added to the selection DAG for:

- Getting a unique seal;
- Sealing the stack capability;
- Shrinking and uninitializing the stack capability;
- Argument spilling (if necessary);
- Creating the return capability and sealing it;
- Clearing non-argument registers.

After the capability jump node another node needs to be added to move the contents of the *IDC* register to the capability stack register. One important requirement is that the stack capability should begin at an address that is a multiple of 32, therefore it might be necessary to insert some store instructions before shrinking and uninitializing the stack capability if the cursor of the stack capability is not currently a multiple of 32. These store instructions should write beneath the cursor (with an offset of -1) until the cursor is a multiple of 32.

8.4 Returning From A Function

Returns need to be done with the `ccall $c1, $c2, 1` instruction. This modification needs to be done in *MipsISelLowering.cpp*, in the *LowerReturn* function. At the end of this function the return node is created. The return node created for the new calling convention needs to be a `ccall` return node. This selection DAG node still needs to be created, this is similar to the current jump nodes and can be done in *MipsInstrCheri.td* (should be defined near the `CapJmpLink` node definition, as it concerns jumps).

8.5 Function Prologue

The function prologue is no longer required to decrement the offset of the stack capability, or preserve callee-saved registers (non-argument registers are cleared before the function is invoked). The prologue should check that the available stack capability has the right permissions. This can be implemented by emitting the machine instructions, i.e. the assembly code, of Section 5.4. Alternatively a pseudo instruction could be defined that represents the instruction sequence to check the stack capability. The function prologue will only have to emit this pseudo instruction at the beginning of the function then. It might be necessary to spill *$c1* and *$c2* to the stack, these are the registers containing the return capability and the caller's stack capability, these will be needed to return to the caller.

These modifications need to be made in *MipsSEFrameLowering.cpp*, in the *emitPrologue* function.

8.6 Function Epilogue

The function epilogue requires a few changes (in the *emitEpilogue* function of *MipsSEFrameLowering.ccp*). The stack frame of the function needs to be cleared, this means overwriting the stack

with zeroes. A loop for this will have to be added to the *emitEpilogue* function (in *MipsSEFrameLowering.cpp*), that writes zeroes over the entire stack frame. Stack frames start at addresses that are multiples of 32 and their size is multiples of 32 too, so we can say that the size of a stack frame is of the form $32x$, where $x = stack_frame_size/32$. There are a few options for clearing the stack (*$c11* is the stack capability):

- For the 256 bit capability format each `ucsc $c11, $cnull, y($c11)` (where y is 0, 1, ..., $x - 1$) will overwrite every 32 bytes with the null capability. This will not write over the stack frame with all zeroes, because the null capability has the otype that are all ones. But no sensitive data or (valid) capabilities will be left on the stack;
- Completely zero out the stack by using `ucsd $c11, $zero, y($c11)` (where y is 0, 1, ..., $(x * 4) - 1$, 32 bytes consist of 4 double words).

After clearing the stack, the non-return registers still need to be cleared, the epilogue needs to emit the corresponding clear instructions discussed in Section 5.5.

In the current epilogue, the stack capability offset is incremented again to pop the stack frame. This is no longer needed because the previous stack capability will be available at the caller's return site.

9 Conclusions

The semantics of the uninitialized capabilities were described at the beginning of the thesis. After describing the semantics of uninitialized capabilities, the ISA extension for uninitialized capabilities was described. This ISA extension was then instantiated for CHERI-MIPS in software (using a simulator). Uninitialized capabilities have not been added in hardware and is out of the scope if this thesis. I do believe it should be possible to implement them in hardware, the reasoning behind this is that uninitialized capabilities only require one bit to indicate if a capability is uninitialized or not, and there are some padding bits available in the 256-bit capability format. Other formats should be possible too but are more complicated due to the compression techniques used. The instruction modifications and new instructions seem plausible in hardware, they have existing counterparts (the normal store instructions and set bounds instruction) and because these could be implemented in hardware, it seems safe to say that the ones introduced by this thesis should be as well.

A modified version of the calling convention with local capabilities, using uninitialized capabilities as well, is described in detail. This calling convention is more secure than existing alternatives but at the cost of some overhead due to stack and register clearing requirements. The assembler of the LLVM project, for the MIPS backend, was modified to support the uninitialized capability instructions. The assembler was then used to evaluate the instructions and the secure calling convention.

In this evaluation the difference in the number of instructions for the original and secure calling convention for a few example programs was discussed. This discussion lead to the observation that using the secure calling convention doubles the number of instructions. The execution time for the example programs was measured and the secure calling convention is slower due to its overhead by clearing registers and having functions clear their stack frames before returning to their caller. The register clearing could be faster in future implementations of CHERI-MIPS by keeping a valid bit per register and clearing this bit instead of zeroing the register [18, page 194]. Optimized programs will have a smaller overhead then their unoptimized versions, this is because stack frames are smaller and this leads to less clearing before a function returns to its caller.

Finally, an exploration of the modifications needed to the CLang/LLVM compiler were outlined and discussed. Although there was not enough time to implement these myself, the exploration should provide useful in future work, be it building on this thesis and finishing the implementation of the calling convention or for adding other calling conventions to the CLang/LLVM compiler.

References

[1] Nicholas P Carter, Stephen W Keckler, and William J Dally. "Hardware support for fast capability-based addressing". In: *ACM SIGOPS Operating Systems Review* 28.5 (1994), pp. 319–327.

[2] Jack B Dennis and Earl C Van Horn. "Programming semantics for multiprogrammed computations". In: *Communications of the ACM* 9.3 (1966), pp. 143–155.

[3] Robert S. Fabry. "Capability-based addressing". In: *Communications of the ACM* 17.7 (1974), pp. 403–412.

[4] Alexandre Joannou et al. "Efficient tagged memory". In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 641–648.

[5] Henry M Levy. *Capability-based computer systems*. Digital Press, 1984.

[6] Kayvan Memarian et al. "Exploring C semantics and pointer provenance". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–32.

[7] James H Morris Jr. "Protection in programming languages". In: *Communications of the ACM* 16.1 (1973), pp. 15–21.

[8] Kyndylan Nienhuis et al. *Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process*. Tech. rep. University of Cambridge, Computer Laboratory, 2019.

[9] Marco Patrignani, Amal Ahmed, and Dave Clarke. "Formal approaches to secure compilation: A survey of fully abstract compilation and related work". In: *ACM Computing Surveys (CSUR)* 51.6 (2019), pp. 1–36.

[10] David A Patterson and John L Hennessy. *Computer Organization and Design: The Hardware Software Interface*. Morgan kaufmann, 2014.

[11] George Pirocanac. *MIPSproTM N32 ABI Handbook*. Ed. by SusanEditor Wilkening. Silicon Graphics, Inc., 2002.

[12] David D Redell. *Naming and protection in extendible operating systems*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1974.

[13] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "Reasoning about a machine with local capabilities". In: *European Symposium on Programming*. Springer. 2018, pp. 475–501.

[14] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–28.

[15] Riccardo Solmi. *MIPS Assembly Language Programmer's Guide*. Silicon Graphics, Inc., 1992f,

[16] Stylianos Tsampas et al. "Towards automatic compartmentalization of C programs on capability machines". In: *Workshop on Foundations of Computer Security 2017*. 2017, pp. 1–14.

[17] Robert NM Watson et al. *Capability Hardware Enhanced RISC Instructions: Cheri Instruction-Set Architecture*. Tech. rep. University of Cambridge, Computer Laboratory, 2014.

[18] Robert NM Watson et al. *Capability Hardware Enhanced RISC Instructions: Cheri Instruction-Set Architecture (Version 7)*. Tech. rep. University of Cambridge, Computer Laboratory, 2019.

[19] Robert NM Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide*. Tech. rep. University of Cambridge, Computer Laboratory, 2015.

[20] Robert NM Watson et al. "Cheri: A hybrid capability-system architecture for scalable software compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 20–37.

[21] Robert NM Watson et al. "Fast protection-domain crossing in the cheri capability-system architecture". In: *IEEE Micro* 36.5 (2016), pp. 38–49.

[22] Jonathan Woodruff et al. "Cheri concentrate: Practical compressed capabilities". In: *IEEE Transactions on Computers* 68.10 (2019), pp. 1455–1469.

A Sail Common Definitions

This appendix describes the Sail functions used in the thesis.

`capToMemBits : Capability -> bits(256)`

`capToMemBits` is the reverse of `memBitsToCapability`.

`checkCP2usable : unit -> unit`

`checkCP2usable` raises a co-processor unusable exception if `CP0Status.CU[2]` is not set. All capability instructions must first check that the capability co-processor is enabled. This allows the operating system to only save and restore the full capability context for processes that use capabilities.

`getCapBase : Capability -> uint64`

returns the base of the given capability as an integer in the range 0 to $2^{64} - 1$.

`getCapCursor : Capability -> uint64`

returns the address of the capability as an integer in the range 0 to $2^{64} - 1$.

`getCapOffset : Capability -> uint64`

returns the offset of the capability (i.e. address relative to base) as an integer in the range 0 to $2^{64} - 1$

`getCapTop : Capability -> CapLen`

returns the top of the given capability as an integer in the range 0 to 2^{65}.

`incCapOffset : (Capability, bits(64)) -> (bool, Capability)`

`incCapOffset` is the same as `setCapOffset` except that the 64-bit value is added to the current capability offset modulo 2^{64} (i.e. signed twos-complement arithemtic).

`isAddressAligned : (bits(64), WordType) -> bool`

`isAddressAligned(address, wordtype)` returns whether `address` is naturally aligned for the given `wordtype`.

`memBitsToCapability : (bool, bits(256)) -> Capability`

`memBitsToCapability(tag, capBits)` converts `capBits` from the in-memory capability format to a convenient structure for ease of field access. Note that the bit representation is xored with the bit representation of the null capability to ensure that null is always stored as all-zeros in memory.

```
MEMw_tagged : forall ('size : Int), 'size > 0. (bits(64), int('size), bool,
↪ bits('size * 8)) -> unit
```

`MEMw_tagged(addr, size, t, value)` writes `size` bytes, `value`, to physical address, `addr`, with associated tag value, `t`.

```
MEMw_wrapper : forall ('n : Int), 'n >= 1. (bits(64), int('n), bits(8 * 'n)) ->
↪  unit
```

`MEMw_wrapper(addr, size, value)` writes `size` bytes of `value` to physical address `addr`.

`ones_implicit : forall ('n : Int), 'n >= 0. (implicit('n), unit) -> bits('n)`

`pow2 : forall ('n : Int), 'n >= 0. int('n) -> int(2 ^ 'n)`

`pow2`(n) returns 2 raised to the power n.

`raise_c2_exception : forall ('o : Type). (CapEx, regno) -> 'o`

causes the processor to raise a capability exception by writing the given capability exception cause and register number to the CapCause register then signalling an exception using `SignalException` (on CHERI-MIPS this is a C2E exception in most cases, or a special C2Trap for CCall and CReturn).

`readCapReg : regno -> Capability`

`readCapReg` reads a given capability register or, the null capabiility if the argument is zero.

`readCapRegDDC : regno -> Capability`

`readCapRegDDC` is the same as `readCapReg` except that when the argument is zero the value of DDC is returned instead of the null capability. This is used for instructions that expect an address, where using null would always generate an exception.

`rGPR : bits(5) -> bits(64)`

Reads the value of the given general purpose register as a 64-bit vector. Register zero is always zero.

`setCapAddr : (Capability, bits(64)) -> (bool, Capability)`

`setCapAddr`(cap, addr) returns a new capability derived from `cap` with the address set to `addr`. If the operation fails due to representability checks then the result will have the expected address but the bounds may be incorrect.

`setCapOffset : (Capability, bits(64)) -> (bool, Capability)`

`setCapOffset`(cap, off) returns a new capability derived from `cap` with the offset set to `off` (i.e. with $address = cap.base + off$). If the operation fails due to representability checks then the result will have the expected address but the bounds may be incorrect. Note that, for performance reasons, an approximate representability check may be used that means the operation could fail even though the result would be representable.

`shrinkCap : (Capability, bits(64), bits(65)) -> (bool, Capability)`

`shrinkCap(cap, base, address)` changes the bounds of the capability by setting the base to the given base and the length of the capability to the given address minus the new base , $base = base$ and $length = address - base$.

`SignalException : forall ('o : Type). Exception -> 'o`

Causes the processor to raise the given exception in the usual manner defined by the processor architecture (as modified for CHERI).

`SignalExceptionBadAddr : forall ('o : Type). (Exception, bits(64)) -> 'o`

causes the processor to raise the given exception as per `SignalException`, but with an associated bad address (on MIPS this is written to the BadVAddr register to aid with exception handling).

```
signed : forall ('n : Int), 'n > 0. bits('n) -> range(- (2 ^ ('n - 1)), 2 ^ ('n
↪  - 1) - 1)
```

converts a bit vector of length n to an integer in the range -2^{n-1} to $2^{n-1}-1$ using twos-complement.

`TLBTranslate : (bits(64), MemAccessType) -> bits(64)`

`TLBTranslate(addr, acces_type)` translates the virtual address, `addr`, to a physical address assuming the given `access_type` (load or store). If the TLB lookup fails an ISA exception is raised.

`TLBTranslateC : (bits(64), MemAccessType) -> (bits(64), MemAccessCapRestriction ↪ )`

`TLBTranslateC` is the same as `TLBTranslate` except that it also returns any constraints on tagged capability operations.

`to_bits : forall ('l : Int), 'l >= 0. (int('l), int) -> bits('l)`

`to_bits(l, v)` converts an integer, `v`, to a bit vector of length `l`. If `v` is negative a twos-complement representation is used. If `v` is too large (or too negative) to fit in the requested length then it is truncated to the least significant bits.

`uninitCap : Capability -> Capability`

`unrepCap : Capability -> Capability`

`unrepCap(cap)` returns `cap` with the tag unset. It is used when the result of a capability operation (e.g. setting the address) has caused it to become unrepresentable. The result of subsequent operations such as `getCapBase` will depend on the exact capability format in use but in general the address, length, object type and permissions will remain the same, with the expected operation having been applied to the address.

`unsigned : forall ('n : Int). bits('n) -> range(0, 2 ^ 'n - 1)`

converts a bit vector of length n to an integer in the range 0 to $2^n - 1$.

`wGPR : (bits(5), bits(64)) -> unit`

`wGPR`(rd, v) writes the 64-bit value, v, to the general purpose register *rd*. Writes to register zero are ignored.

`wordWidthBytes : WordType -> range(1, 8)`

Returns the width of the given WordType (byte, half, word, double) in bytes.

`writeCapReg : (regno, Capability) -> unit`

`writeCapReg(cd, cap_val)` writes capability, `cap_val` capability register `cd`. Writes to register zero are ignored.

`zeros_implicit : forall ('n : Int), 'n >= 0. (implicit('n), unit) -> bits('n)`