UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

Research project report

Candidate **2095J**

Project Title

Capability-Based Memory Protection for Scalable Vector Processing

Submitted in partial fulfilment of the requirements for the

MPhil in Advanced Computer Science

Postgraduate Education Office, Room FS03
Department of Computer Science and Technology, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD

Total Page Count: 107

Main chapters (excluding front-matter, references, appendix): 49 (pp 13-61)

Main chapters word count: 14,977 words

Word count methodology: `./texcount.pl -1 -sum -merge -q chapters.tex`
TeXcount version: 3.2.0.41
Counts body text, heading text, and caption text. Does not include text in tables, figures, code listings, or appendices. The anonymous version redacts some trivially identifying information, but uses the same word count as the de-anonymized version. The appendices contain non-essential data that is summarized in the main text (e.g. full results, details on how to compile code), or re-statements of work done in the main text (e.g. a summary of all changes made to the CHERI-RISC-V spec).

Abstract

Capability-Based Memory Protection for Scalable Vector Processing

CHERI, a generic architecture extension which improves memory safety, has garnered attention from industry partners for its low overhead and compatibility with existing source code. CHERI has been adapted to multiple ISAs, including RISC-V and Arm, but not to any scalable vector processors.

Vector processing, where the same operation is performed on multiple elements of a "vector" in parallel, is used everywhere in modern computing from high-performance number-crunching to the humble `memcpy`. Arm SVE[1] and RISC-V "V"[2] are new flagship vector extensions for Arm and RISC-V, which use a "vector-length agnostic programming model" to allow hardware implementations to choose their vector lengths. These scalable vector models are intended to stay in use long into the future, and it is essential for CHERI to support them.

This dissertation focuses on RISC-V V, presenting and evaluating a possible "CHERI-RVV" combination ISA by building and testing a reference implementation in Rust. We find that RVV is easily adaptable to CHERI with no issues, even maintaining binary compatibility with vanilla RVV programs, although other models like Arm SVE may require more investigation. We find a set of issues with the current CHERI compiler that make source-level compatibility difficult, and show they can be easily resolved with engineering effort. Finally, we explore storing capabilities-in-vectors in a limited context, to allow implementing `memcpy` with vector instructions, and show it does not violate security properties.

We conclude that it is viable to combine RVV with CHERI to enable vectorized arithmetic and `memcpy` operations without sacrificing performance, source-level compatibility, or memory protection.

Contents

CHAPTER 1

Introduction

Since 2010, the Cambridge Computer Lab (in association with SRI) has been developing the CHERI[1] architecture extension, which improves the security of any given architecture by checking all memory accesses in hardware. The core impact of CHERI, on a hardware level, is that memory can no longer be accessed directly through raw addresses, but must pass through a *capability*[3]. Capabilities are unforgeable tokens that grant fine-grained access to ranges of memory. Instead of generating them from scratch, capabilities must be *derived* from another capability with greater permissions. For example, a capability giving read-write access to an array of structures can be used to create a sub-capability granting read-only access to a single element. This vastly reduces the scope of security violations through spatial errors (e.g. buffer overflows[4]), and creates interesting opportunities for software compartmentalization[5].

Industry leaders have recognized the value CHERI provides. Arm Inc have manufactured the Morello System-on-Chip, based on their Neoverse N1 CPU, which incorporates CHERI capabilities into the Armv8.2 ISA. While this represents a great step forward, there are still elements on the SoC that haven't fully embraced CHERI (e.g. the GPU), and architecture extensions that haven't been investigated in the context of CHERI. One such example is Arm's Scalable Vector Extension (introduced in Armv8.2 but not included in Neoverse N1), which is designed to remain in use well into the future[1]. Supporting this and other scalable vector ISAs in CHERI is essential to CHERI's long-term relevance.

In the context of modern computer architecture, vector processing is the practice of dividing a large hardware register into a *vector* of multiple *elements* and executing the same operation on each element in a single instruction[2]. This data-level parallelism can drastically increase throughput, particularly for arithmetic-heavy programs. However, before computing arithmetic, the vectors must be populated with data.

[1] Capability Hardware Enhanced RISC Instructions

[2] This is a SIMD (Single Instruction Multiple Data) paradigm.

1.1 Motivation

Modern vector implementations all provide vector load/store instructions to access a whole vector's worth of memory. These range from simple contiguous accesses (where all elements are next to each other), to complex indexed accesses (where each element loads from a different location based on another vector). They can also have per-element semantics, e.g. "elements must be loaded in order, so if one element fails the preceding elements are still valid"[2, Section 7.7]. If CHERI CPUs want to benefit from vector processing's increased performance and throughput, they must support those instructions at some level. But adding CHERI's bounds-checking to the mix may affect these semantics, and could impact performance (e.g. checking each element's access in turn may be slow).

Vector memory access performance is more critical than one may initially assume, because vectors are used for more than just computation. A prime example is `memcpy`: for `x86_64`, `glibc` includes multiple versions of the function[3] taking advantage of vector platforms, then selects one to use at runtime[4]. These implementations are written in assembly and heavily optimized. If the memory accesses are hitting the cache, a few extra cycles of bounds-checking for each access could actually make a noticeable difference.

`memcpy` also raises the important question of how the vector model interacts with capabilities. In non-CHERI processors, `memcpy` will copy pointers around in memory without fuss. For a CHERI-enabled vector processor to support this, it would need to be able to load/store capabilities from vectors without violating any security guarantees. This may require more guarantees than otherwise necessary - for example, each vector register likely needs to be as large or larger than a single capability.

To explore this topic, we chose to focus on the RISC-V Vector extension[2] (shortened to RVV throughout). As of November 2021 this has been ratified by RISC-V International[5], and will be RISC-V's standard vector instruction set moving forward. This has two key benefits. Studying RVV will allow reference "CHERI-RVV" implementations to be built for the CHERI project's open-source RISC-V cores[6], which don't currently support vector processing. Secondly, RVV is a *scalable* vector model, where the length of each vector is implementation-dependent. This has more potential roadblocks than a fixed-length vector model, and investigating them here will make life easier if Arm wish to integrate their Scalable Vector Extension with CHERI later down the road.

[3] It appears memcpy is implemented as a copy of memmove.
[4] `sysdeps/x86_64/multiarch/ifunc-memmove.h in bminor/glibc on GitHub`
[5] `https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions`
[6] `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html`

1.2 Hypotheses and Aims

The goal of this project is to investigate the impact of, and the roadblocks for, integrating a scalable vector architecture with CHERI's memory protection system. In particular, we focus on integrating RVV with the CHERI-RISC-V ISA, with the aim of enabling a future CHERI-RVV implementation and informing the approach for a future CHERI Arm SVE implementation.

The investigation was carried out by designing and testing a CHERI-RVV emulator written in Rust, but that is only a single implementation. To show that RVV can be integrated with CHERI-RISC-V for a wide range of processors, we use information gathered from the emulator to check nine hypotheses (Table 1.1). Hypotheses H-1 and H-2 consider basic feasibility and potential hardware performance issues. Hypotheses H-3 to H-6 ensure that vector software is compatible with potential CHERI-RVV software stacks. Hypotheses H-7 to H-9 considers capabilities-in-vectors: the conditions under which vector registers can hold capabilities, and vectorized instructions can manipulate them.

This document also examines current practical realities, such as the limitations of the current CHERI-RVV software stack, and explains how to compile/execute CHERI-RVV code on baremetal platforms. The RVV and CHERI-RISC-V specifications are succinctly explained in Chapter 2.

Hardware Hypotheses — Chapter 3	
H-1	It is possible to use CHERI capabilities as memory references in all vector instructions.
H-2	The capability bounds checks for vector elements within a known range (e.g. a cache line) can be performed in a single check, amortizing the cost.
Software Hypotheses — Chapter 4	
H-3	Vector code can be compiled in legacy forms (with integer addressing) and function correctly on CHERI with no source code changes.
H-4	Legacy vector code can be compiled into a pure-capability form with no changes.
H-5	Vector code that saves/restores variable-length vectors to/from the stack can be compiled on CHERI-RVV with no source code changes.
H-6	CHERI-vector code can run correctly in multiprocessing systems, where execution may be paused and resumed on interrupts or context switches.
Capabilities-in-Vectors — Chapter 5	
H-7	It is possible for vector registers to hold capabilities to enable copying without violating CHERI security principles.
H-8	It is possible for vector memory accesses to load and store capabilities from vector registers without violating CHERI security principles.
H-9	It is possible for vector instructions to manipulate capabilities in vector registers without violating CHERI security principles.

Table 1.1: Project Hypotheses

CHAPTER 2

Background

This chapter describes RISC-V (Section 2.1), RVV (Sections 2.2 to 2.4), and CHERI (Section 2.6) to the detail required to understand the rest of the dissertation. It summarizes the relevant sections of the RISC-V unprivileged spec[6], the RISC-V "V" extension specification v1.0[2, Sections 1–9, 17], the TR-951 CHERI ISAv8 technical report[7, Chapters 5, 8], and the TR-949 technical report about C/C++ safety on CHERI[8, Section 4.4, Appendix C]. Both vectors and CHERI are described, because this dissertation caters to those who may be familiar with one but not the other.

2.1 RISC-V

RISC-V is an open family of ISAs which defines "base integer ISAs" (e.g. all 64-bit RISC-V cores implement the RV64I base ISA) and extensions (e.g. the "M" extension for integer multiplication). A base instruction set combined with a set of extensions is known as a RISC-V ISA. Because RISC-V is open, anyone can design, manufacture, and sell chips implementing any RISC-V ISA.

Each RISC-V implementation has a set of constant parameters. The most common example is `XLEN`, the length of an integer register in bits, which is tied to the base integer ISA (e.g. 64-bit ISA implies `XLEN=64`). Other constant parameters include `CLEN`, the length of a capability in bits, defined by CHERI relative to `XLEN`; and `VLEN` and `ELEN`, which are used by RVV and entirely implementation-defined.

The extensions of most relevance to this project are the "V" vector extension (RVV, specified in [2]) and the CHERI extension (specified in [7]). RVV has recently been officially ratified, and is the de facto vector extension for RISC-V. The following sections summarize the vector extension, how it accesses memory, and previous implementations in academia.

2.2 A brief history of vector processing

Many vector implementations (Intel SSE/AVX, Arm's Advanced SIMD and Neon) use fixed-length vectors - e.g. 128-bit vectors which a program interprets as four 32-bit elements. As the industry's desire for parallelism grew, new implementations had to be designed with longer vectors of more elements. For example, Intel SSE/SSE2 (both 128-bit) was succeeded by AVX (128 and 256-bit), then AVX2 (entirely 256-bit), then AVX-512 (512-bit). Programs built for one extension, and hence designed for a specific vector size, could not automatically take advantage of longer vectors.

Scalable vectors address this by not specifying the vector length, and instead calculating it on the fly. Instead of hardcoding "this loop iteration uses a single vector of four 32-bit elements", the program has to ask "how many 32-bit elements will this iteration use?". This gives hardware designers more freedom, letting them select a suitable hardware vector length for their power/timing targets, while guaranteeing consistent execution of programs on arbitrarily-sized vectors. RVV uses a scalable vector model.

2.3 The RVV vector model

Summarizes [2, Sections 1-6, 17]

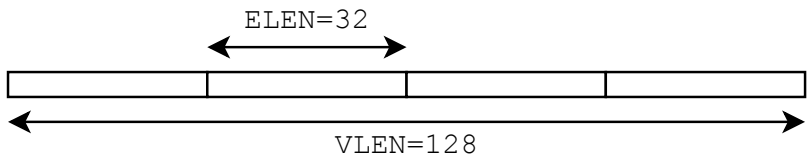


Figure 2.1: Example of a RVV vector register
`VLEN = 128, ELEN = 32`

RVV defines thirty-two vector registers, each of an implementation-defined constant width `VLEN`. These registers can be interpreted as *vectors* of *elements*. The program can configure the size of elements, and the implementation defines a maximum width `ELEN`. Fig. 2.1 shows a simple example.

RVV also adds some state that defines how the vector registers are used (see Fig. 2.2). These are stored in RISC-V Control and Status Registers (CSRs), which the program can read. `vtype` (Section 2.3.1) defines how the vector registers are split into elements. `vstart` and `vl` (Section 2.3.2) divides the elements into three disjoint subsets: *prestart*, the *body*, and the *tail*. Masked accesses (Section 2.3.3) further divide the *body* into *active* and *inactive* elements. This section also describes the vector exception model (Section 2.3.4).

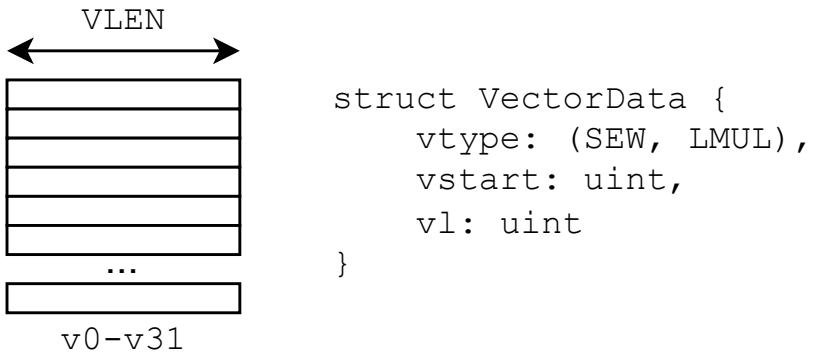


Figure 2.2: Summary of additional state used by RVV

2.3.1 `vtype`

The `vtype` CSR contains two key fields that describe how vector instructions interpret the contents of vector registers. The first is the Selected Element Width (`SEW`), which is self-explanatory. It can be 8, 16, 32, or 64. 128-bit elements are referenced a few times throughout but haven't been formally specified (see [2, p10, p32]).

The second field is the Vector Register Group Multiplier (`LMUL`). Vector instructions don't just operate over a single register, but over a register *group* as defined by this field. For example, if `LMUL=8` then each instruction would operate over 8 register's worth of elements. These groups must use aligned register indices, so if `LMUL=4` all vector register operands should be multiples of 4 e.g. `v0`, `v4`, `v8` etc. In some implementations this may increase throughput, which by itself is beneficial for applications.

However, the true utility of `LMUL` lies in widening/narrowing operations (see Fig. 2.3). For example, an 8-by-8-bit multiplication can produce 16-bit results. Because the element size doubles, the number of vector registers required to hold the same number of elements also doubles. Doubling `LMUL` after such an operation allows subsequent instructions to handle all the results at once. At the start of such an operation, fractional `LMUL` (1/2, 1/4, or 1/8) can be used to avoid subsequent results using too many registers.

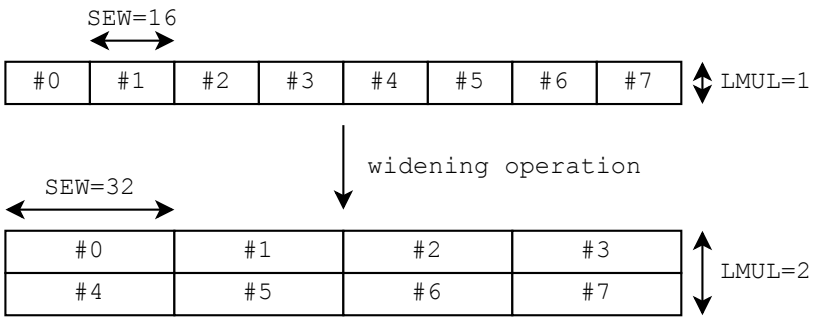
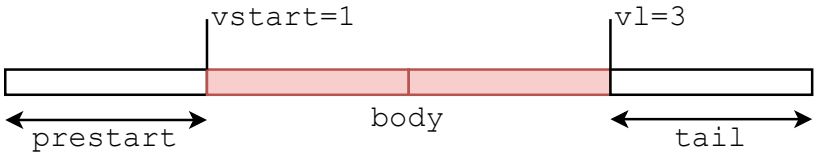


Figure 2.3: Example of using LMUL to access results of widening operations

(a) Fully utilized vector



(b) Partially utilized vector

Figure 2.4: Examples of vector utilization with `vl` and `vstart`

`vtype` also encodes two flags: mask-agnostic and tail-agnostic. If these are set, the implementation is *allowed* to overwrite any masked-out or tail elements with all 1s.

Most vector instructions will interpret their operands using `vtype`, but this is not always the case. Some instructions (such as memory accesses) use different Effective Element Widths (`EEW`) and Effective LMULs (`EMUL`) for their operands. In the case of memory accesses, the `EEW` is encoded in the instruction bits and the `EMUL` is calculated to keep the number of elements consistent. Another example is widening/narrowing operations, which by definition have to interpret the destination registers differently from the sources.

Programs update `vtype` through the `vsetvl` family of instructions. These are designed for a "stripmining" paradigm, where each iteration of a loop processes some elements until all elements are processed. `vsetvl` instructions take a requested `vtype` and the number of remaining elements to process (the Application Vector Length or `AVL`), and return the number of elements that will be processed in this iteration. This value is saved in a register for the program to use, and also saved in the internal `vl` CSR.

2.3.2 `vl` and `vstart` — Prestart, body, tail

The first CSR is the Vector Length `vl`, which holds the number of elements that could be updated from a vector instruction. The program updates this value through fault-only-first loads (Section 2.5.2) and more commonly `vsetvl` instructions.

In the simple case, `vl` is equal to the total available elements (see Fig. 2.4a). It can also be fewer (see Fig. 2.4b), in which case vector instructions will not write to elements in the "tail" (i.e. elements past `vl`). This eliminates the need for a 'cleanup loop' common in fixed-length vector programs.

In a similar vein, `vstart` specifies "the index of the first element to be executed by a vector instruction". Elements before `vstart` are known as the *pre-start* and are not touched by executed instructions. It is usually only set by the hardware whenever it is interrupted mid-instruction (see Fig. 2.5 and Section 2.3.4) so that the instruction can be re-executed later

without corrupting completed values. Whenever a vector instruction completes, `vstart` is reset to zero.

The program *can* set `vstart` manually, but it may not always work. If an implementation couldn't arrive at the value itself, then it is allowed to reject it. The specification gives an example where a vector implementation never takes interrupts during an arithmetic instruction, so it would never set `vstart` during an arithmetic instruction, so it could raise an exception if `vstart` was nonzero for an arithmetic instruction.

2.3.3 Masking — Active/inactive elements

Most vector instructions allow for per-element *masking* (see Fig. 2.6). When masking is enabled, register `v0` acts as the 'mask register', where each bit corresponds to an element in the vector[1]. If the mask bit is 0, that element is *active* and will be used as normal. If the mask bit is 1, that element will be *inactive* and not written to (or depending on the mask-agnostic setting, overwritten with 1s). When masking is disabled, all elements are *active*.

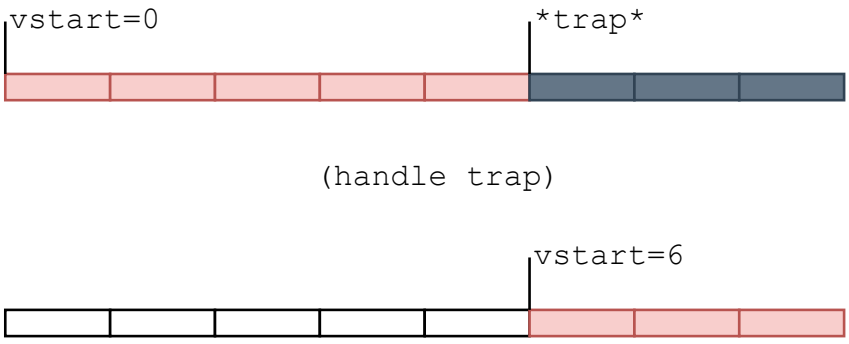


Figure 2.5: Example of the hardware setting `vstart` after a trap

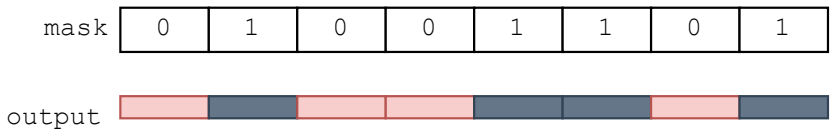


Figure 2.6: Example of masking a vector operation

[1] A single vector register will always have enough bits for all elements. The maximum element count is found when SEW is minimized (8 bits) and LMUL is maximized (8 registers), and is equal to `VLEN * LMUL / SEW = VLEN * 8 / 8 = VLEN`.

2.3.4 Exception handling

Summarizes [2, Section 17]

During the execution of a vector instruction, two events can prevent an instruction from fully completing: a synchronous exception in the instruction itself, or an asynchronous interrupt from another part of the system. Implementations may choose to wait until an instruction fully completes before handling asynchronous interrupts, making it unnecessary to pause the instruction halfway through, but synchronous exceptions cannot be avoided in this way (particularly where page fault exceptions must be handled transparently).

The RVV specification defines two modes for 'trapping' these events, which implementations may choose between depending on the context (e.g. the offending instruction), and notes two further modes which may be used in further extensions. All modes start by saving the PC of the trapping instruction to a CSR `*epc`.

2.3.4.1 Imprecise vector traps

Imprecise traps are intended for events that are not recoverable, where "reporting an error and terminating execution is the appropriate response". They do not impose any extra requirements on the implementation. For example, an implementation that executes instructions out-of-order does not need to guarantee that instructions older than `*epc` have completed, and is allowed to have completed instructions newer than `*epc`.

If the trap was triggered by a synchronous exception, the `vstart` CSR must be updated with the element that caused it. The specification calls out synchronous exceptions in particular, but does not mention asynchronous interrupts. It's likely that imprecise traps for asynchronous interrupts should also set `vstart`, but this issue has been raised with the authors for further clarification[2]. The specification also states "There is no support for imprecise traps in the current standard extensions", meaning that the other standard RISC-V exceptions do not use and have not considered imprecise traps.

2.3.4.2 Precise vector traps

Precise vector traps are intended for instructions that can be resumed after handling the interrupting event. This means the architectural state (i.e. register values) when starting the trap could be saved and reloaded before continuing execution. Therefore it must look like instructions were completed in-order, even if the implementation is out-of-order:

- Instructions older than `*epc` must have completed (committed all results to the architectural state)
- Instructions newer than `*epc` must **not** have altered architectural state.

[2] **(redacted for anonymity)**

On a precise trap, regardless of what caused it, the `vstart` CSR must be set to the element index on which the trap was taken. The save-and-reload expectation then add two constraints on the trapping instruction's execution:

- Operations affecting elements preceding `vstart` must have committed their results
- Operations affecting elements at or following `vstart` must either
  - not have committed results or otherwise affected architectural state
  - be *idempotent* i.e. produce exactly the same result when repeated.

The idempotency option gives implementations a lot of leeway. Some instructions, such as indexed segment loads (Section 2.5.3), are specifically prohibited from overwriting their inputs to make them idempotent. If an instruction is idempotent, an implementation is even allowed to repeat operations on elements *preceding* `vstart`. However for memory accesses the idempotency depends on the memory being accessed. For example, reading or writing a memory-mapped I/O region may not be idempotent.

Another memory-specific issue is that of *demand-paging*, where the OS needs to step in and move virtual memory pages into physical memory for an instruction to use. This use-case is specifically called out by the specification for precise traps. Usually, this is triggered by some element of a vector memory access raising a synchronous exception, invoking a precise trap, and writing the "Machine Trap Value" scalar register with the offending address[9, Section 3.1.21]. `vstart` must be set to an element at (or before[3]) the one that demanded the page, because that element must perform the access after reloading. If an implementation sets `vstart` to the offending element, because operations preceding `vstart` must have completed, any elements that could potentially trigger demand-paging *must* wait for the preceding elements to complete.

2.3.4.3 Other modes

The RVV spec notes two other possible future trap modes. First is "Selectable precise/imprecise traps", where an implementation allows the user to select precise or imprecise traps for e.g. debugging or performance.

The second mode is "Swappable traps", where a trap handler could use special instructions to "save and restore the vector unit microarchitectural state". The intent seems to be to support context switching with imprecise traps, which could also require the *opaque* state (i.e. internal state not visible to the program) to be saved and restored. Right now, it seems that context switching always requires a precise trap.

[3] If the memory region is idempotent, then `vstart` could any value where all preceding elements had completed. It could even be zero, in which case all accesses would be retried on resume, as long as it could guarantee forward progress.

2.3.5 Summary

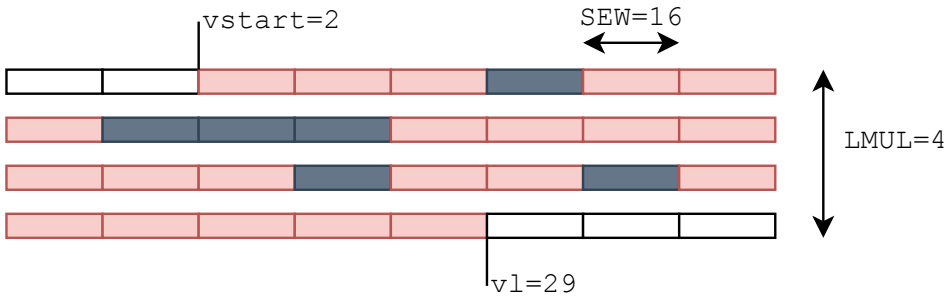


Figure 2.7: Combined examples for RVV vector

Fig. 2.7 shows all of the above features used in a single configuration:

- The instruction was previously interrupted with a precise trap and restarted, so `vstart=2`
- Elements are 16-bit
- `LMUL=4` to try and increase throughput
- Only 29 of the 32 available elements were requested, so `vl=29` (3 tail elements)
- Some elements are masked out/inactive (in this case seemingly at random)
- Overall, 21 elements are active

2.4 Previous RVV implementations

Academia and industry have implemented RVV even before v1.0 was released. The scalable vector model allows great diversity: Johns and Kazmierski integrated a minimal vector processor into a microcontroller's scalar pipeline (`VLEN=32`) [10], Di Mascio et al. used RVV for deep learning in space[11], and AndesCode, SiFive, and Alibaba have released cores with `VLEN`s up to 512[12][13][14]. Other academic examples include Ara[15], Arrow[16], RISC-V^2[17], and Vicuna[18], which all decouple the vector processing from the scalar pipeline.

Very recently, more implementations were revealed at RISC-V Week in Paris (May 2022). Vitruvius[19] uses extremely long vectors `VLEN = 16384`, is implemented as a decoupled processor, and is the first RISC-V processor to support the Open Vector Interface (OVI)[4] to communicate with the scalar core. VecProM[20] splits its approach into two, where vectors beyond a certain length are strip-mined and processed in hardware using a scratch memory, using OVI to connect multiple heterogeneous vector processors to a scalar core. Both were produced from the Barcelona Supercomputing Center under the European Processor Initiative. It seems that adoption of RVV will continue, making it a good choice for adapting to CHERI.

[4] `semidynamics/OpenVectorInterface` on Github

2.5 RVV memory instructions

Summarizes [2, Sections 7-9]

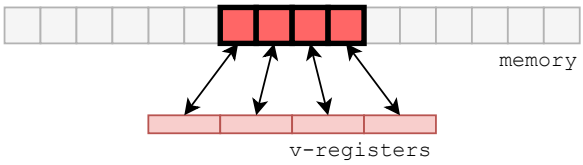
RVV defines three broad categories of memory access instructions, which can be further split into five archetypes with different semantics. This section summarizes each archetype, their semantics, their assembly mnemonics, and demonstrates how they map memory accesses to vector elements.

For the most part, memory access instructions handle their operands as described in Section 2.3. `EEW` and `EMUL` are usually derived from the instruction encoding, rather than reading the `vtype` CSR. In a few cases the Effective Vector Length `EVL` is different from the `vl` CSR, so for simplicity all instructions are described in terms of `EVL`.

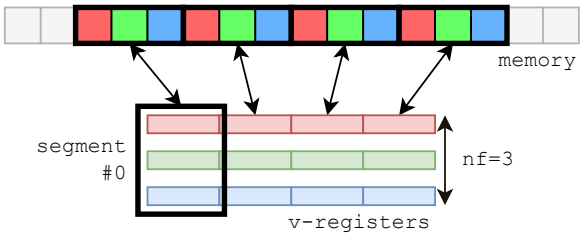
Segmented accesses

Three of the five archetypes (unit/strided, fault-only-first, and indexed) support *segmented* access. This is used for unpacking contiguous structures of $1 \leq \texttt{nf} \leq 8$ *fields* and placing each field in a separate vector. In these instructions, the values of `vl`, `vstart`, and the mask register are interpreted in terms of segments.

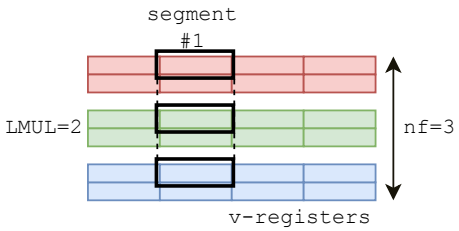
Fig. 2.8 demonstrates a common example: the extraction of separate R, G, and B components from a color. Without segmentation, i.e. $n = 1$, each consecutive memory address maps to a consecutive element in a single vector register group. With segmentation, elements are grouped into segments of $n > 1$ fields, where each field is mapped to a different vector register group. This principle extends to `LMUL > 1` (Fig. 2.8c).



(a) Simple vector element to address mapping



(b) Element-address mapping for segmented access



(c) Example of segment mapping for `LMUL > 1`

Figure 2.8: Comparison between segmented and unsegmented accesses
For readability, the vector registers are 2x as wide

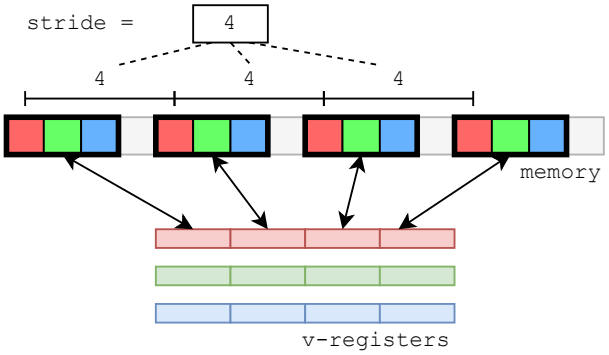
2.5.1 Unit and Strided accesses

	Mnmemonic	Data	Address	Stride	Masked
Unit	`vlseg<nf>e<eew>.v`	`vd,`	`(rs1),`	*implicit*	`vm`
Strided	`vlsseg<nf>e<eew>.v`	`vd,`	`(rs1),`	`rs2,`	`vm`

(a) Instruction

Masked?	`vm == 0`
EEW	`<eew>`
EVL	`vl`
EMUL	`VLEN * <eew> / EVL`
NFIELDS	`<nf>`

(b) Parameters



(c) Example of a segmented strided access `EEW=8-bits`, `nf=3`, `stride=4`,`EVL=4`

Figure 2.9: Segmented Unit/Strided Access Information

This archetype moves active elements of `nf` vector register groups to/from contiguous segments of memory, where the start of each segment is separated by `stride` bytes.

- Unit-stride instructions tightly pack segments, i.e. `stride = nf * eew / 8`.
- Strided instructions read `stride` from register `rs2`.
  - `stride` may be positive, negative, or zero.
- These instructions don't do anything if `vstart >= EVL`.

Strided accesses where `rs2` is register `x0` may perform fewer than `EVL` memory accesses. Otherwise, even if `stride = 0`, implementations must appear to perform all accesses.

Ordering

There are no ordering guarantees, other than those required by precise vector traps (if used).

Exception Handling

If any element within segment i triggers a synchronous exception, `vstart` is set to i and a precise or imprecise trap is triggered. Load instructions may overwrite active segments past the segment index at which the trap is reported, but not past `EVL`.[2, Section 7.7] Upon entering a trap, it is implementation-defined how much of the faulting segment's accesses are performed.

2.5.2 Unit fault-only-first loads

Mnemonic	Data	Address	Masked
`vlseg<nf>e<eew>ff.v`	`vd,`	`(rs1),`	`vm`

(a) Instruction

Masked?	`vm == 0`
EEW	`<eew>`
EVL	`vl`
EMUL	`VLEN * <eew> / EVL`
NFIELDS	`<nf>`

(b) Parameters

Figure 2.10: Unit Fault-only-First Information

This archetype is equivalent to a unit load in all respects but exception handling. If any access in segment 0 raises an exception[5], `vl` is not modified and the trap is taken as usual. If any access in any active segment > 0 raises an exception, the trap is not taken, `vl` is reduced to the index of the offending segment, and the instruction finishes. If an asynchronous interrupt is encountered at any point, the trap is taken and `vstart` is set as usual.

This archetype is intended for "loops with data-dependent exit conditions", and is commonly used for string operations. The specification uses it in a `strcmp` example ([2, Section A.9]).

Similar to plain loads, if an exception is encountered the instruction is allowed to update segments past the offender (but not past the original `vl`). If any synchronous exception or asynchronous interrupt occurs, regardless of the segment index, it is implementation-defined how much of the faulting segment's accesses are performed.

[5]Segment 0 may be masked out, in which case this is impossible.

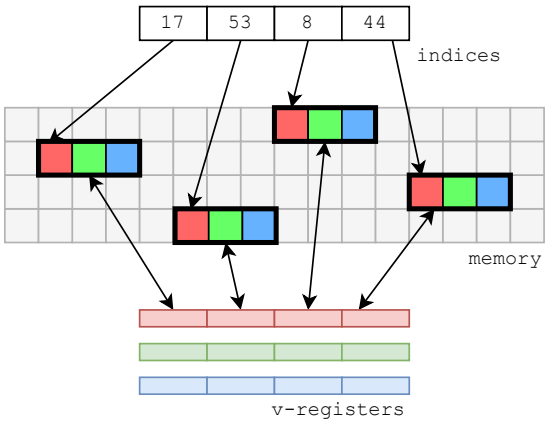
2.5.3 Indexed accesses

Mnmemonic	Data	Address	Indices	Masked
`vl<u|o>xseg<nf>e<eew>.v`	`vd,`	`(rs1),`	`vs2,`	`vm`

(a) Instruction

Masked?	`vm == 0`
Element EEW	`vtype.SEW`
Element EMUL	`vtype.LMUL`
Ordered?	`<u|o>`
Index Vector	`vs2`
Index EEW	`<eew>`
Index EMUL	`VLEN * <eew> / EVL`
NFIELDS	`<nf>`
EVL	`vl`

(b) Parameters



(c) Example of a segmented indexed access `EEW=8-bits`, `nf=3`

Figure 2.11: Segmented Indexed Access Information

This archetype moves elements of `nf` vector register groups to/from contiguous segments of memory, where each segment is offset by an index (in bytes) taken from another vector.

- The start of each segment is defined by `address + index_vector[i]`.
- These instructions don't do anything if `vstart >= EVL`.
- Indexed segment loads may not overwrite the index vector[6].

Ordering

Accesses within each segment are not ordered relative to each other. If the ordered variant of this instruction is used, then the segments must be accessed in order (i.e. 17, 53, 8, 44 for Fig. 2.11c). Otherwise, segment ordering is not guaranteed.

Exception Handling

If any element within segment i triggers a synchronous exception, `vstart` is set to i and a precise or imprecise trap is triggered. Load instructions may overwrite active segments past the segment index at which the trap is reported, but not past `EVL`[2, Section 7.7]. Upon entering a trap, it is implementation-defined how much of the faulting segment's accesses are performed.

[6]This allows restarting after raising an exception partway through a structure

2.5.4 Unit whole-register accesses

Mnemonic	Data	Address
vl<nreg>re<eew>.v	vd,	(rs1)

(a) Instruction

Masked?	False
Registers	<nreg>
EEW	<eew>
EVL	NFIELDS * VLEN / EEW
EMUL	1

(b) Parameters

Figure 2.12: Unit Whole Register Information

This archetype moves the contents of nreg vector registers to/from a contiguous range in memory. Equivalent to a unit-stride access where EVL equals the total number of elements in nreg registers.

- nreg must be a power of two.
- These instructions don't support segmented access.
- These instructions don't do anything if vstart >= EVL.

Ordering and exception handling are identical to unit-stride accesses (Section 2.5.1).

2.5.5 Unit bytemask accesses

Mnemonic	Data	Address
vlm.v	vd,	(rs1)

(a) Instruction

Masked?	False
EEW	8-bits
EVL	ceil(vl/8)
EMUL	1

(b) Parameters

Figure 2.13: Unit Bytemask Information

This archetype moves the contents of a mask register to/from a contiguous range of memory. It transfers at least vl bits, one bit for each element that could be used in subsequent vector instructions. This will always fit in a single vector register (see Section 2.3.3), hence EMUL = 1 in all cases.

- These instructions operate as if the tail-agnostic setting of vtype is true.
- These instructions don't support segmented access.
- These instructions don't do anything if vstart >= EVL.

Ordering and exception handling are identical to unit-stride accesses (Section 2.5.1).

2.6 CHERI

In CHERI, addresses/pointers are replaced with capabilities: unforgeable tokens that provide *specific kinds of access* to an *address* within a *range of memory*. The above statement is enough to understand what capabilities contain[7]:

- Permission bits, to restrict access
- The *cursor*, i.e. the address it currently points to
- The *bounds*, i.e. the range of addresses this capability could point to

A great deal of work has gone into compressing capabilities down into a reasonable size (see [21], Fig. 2.14), and using the magic of floating-point all of this data has been reduced to just 2x the architectural register size. For example, on 64-bit RISC-V a standard capability is 128-bits long. The rest of this dissertation assumes capabilities are 128-bits long for simplicity.

A CHERI implementation has to enforce three security properties about its capabilities[7, Section 1.2.1]:

- Provenance — Capabilities must always be derived from valid manipulations of other capabilities.
- Integrity — Corrupted capabilities cannot be dereferenced.
- Monotonicity — Capabilities cannot increase their rights.

Integrity is enforced by tagging registers and memory. Every 128-bit register and aligned 128-bit region of memory has an associated tag bit, which denotes if its data encodes a valid capability[8]. If any non-capability data is written to any part of the region the tag bit is zeroed out. Instructions that perform memory accesses can only do so if the provided capability has a valid tag bit. As above, significant work has gone into the implementation to reduce the DRAM overhead of this method (see [22]).

[7] This is a slight simplification. For the purposes of vector memory accesses the *otype* of a capability can be ignored, as any type other than `UNSEALED` cannot be dereferenced anyway.

[8] This has the side-effect that capabilities must be 128-bit aligned in memory.

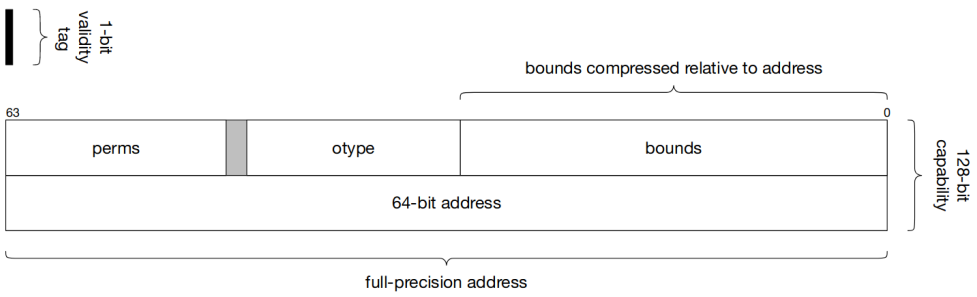


Figure 2.14: 128-bit compressed capability representation — from [3]

Provenance and Monotonicity are enforced by all instructions that manipulate capabilities. If an implementation detects a violation of either property, it will zero out the tag bit and rely on Integrity enforcement to ensure it is not dereferenced. Some CHERI-enabled architectures, such as CHERI-RISC-V, also raise a synchronous exception when this occurs.

2.6.1 CHERI-RISC-V ISA

The Cambridge Computer Lab's TR-951 report[7] describes the latest version of the CHERI architecture (CHERI ISAv8) and proposes applications to MIPS, x86-64, and RISC-V. CHERI-RISC-V is a mostly straightforward set of additions to basic RISC-V ISAs. It adds thirty-two general-purpose capability registers, thirty-two Special Capability Registers (SCRs), and many new instructions.

The new general-purpose capability registers are each of size `CLEN = 2 * XLEN` plus a tag bit. These registers store compressed capabilities. While there is always a logical distinction between the pre-existing *integer* registers `x0-x31` and the *capability* registers `cx0-cx31`, the architecture may store them in a Split or Merged register file. A Split register file stores the integer registers separately from capability registers, so programs can manipulate them independently. A Merged register file stores thirty-two registers of length `CLEN`, using the full width for the capability registers, and aliases the integer registers to the bottom `XLEN` bits. Under a merged register file, writing to an integer register makes the capability counterpart invalid, so programs have to be more careful with register usage.

Many of the new SCRs are intended to support the privileged ISA extensions for e.g. hypervisors or operating systems. The emulator doesn't use these, so their SCRs are not listed here, but there are two highly relevant SCRs for all modes: the Program Counter Capability and the Default Data Capability.

The PCC replaces the program counter and adds more metadata, ensuring instruction fetches have the same security properties as normal loads and stores. The DDC is used to sandbox integer addressing modes. CHERI-RISC-V includes new instructions which use integer addressing, and allows legacy (i.e. integer addressed) code to function on CHERI systems without recompiling for CHERI-RISC-V. These instructions all use integer addresses relative to the DDC, and the DDC controls the permissions those instructions have.

2.6.2 Instruction changes

TR-951[7, Chapter 8] specifies a suite of new instructions, as well as a set of modifications to pre-existing instructions. Many of the new instructions are unrelated to pre-existing instructions, and implement capability-specific operations like accessing fields of capability registers. The most relevant new instructions for our case are the various loads/stores.

Name	Direction	Data type	Address calculation
L[BHWD][U].CAP	Load	Integer	via capability register
L[BHWD][U].DDC	Load	Integer	via DDC
LC.CAP	Load	Capability	via capability register
LC.DDC	Load	Capability	via DDC
S[BHWD].CAP	Store	Integer	via capability register
S[BHWD].DDC	Store	Integer	via DDC
SC.CAP	Store	Capability	via capability register
SC.DDC	Store	Capability	via DDC

Table 2.1: New CHERI load/store instructions

Name	Direction	Data type	Address calculation (Capability/Integer mode)
[C]LC[1]	Load	Capability	via capability/DDC
[C]SC[2]	Store	Capability	via capability/DDC
L[BWHD][U]	Load	Integer	via capability/DDC
S[BWHD]	Store	Integer	via capability/DDC
FL[WDQ]	Load	Float	via capability/DDC
FS[WDQ]	Store	Float	via capability/DDC
LR	Load	Integer	via capability/DDC
SC	Store	Integer	via capability/DDC
AMO[3]	—	Integer	via capability/DDC

[1] Replaces RV128 LQ [2] Replaces RV128 SQ [3] All atomic memory operations

Table 2.2: Preexisting RISC-V load/store instructions modified by CHERI-RISC-V

CHERI-RISC-V adds new instructions for loading integer and capability data, either via capabilities or using integer addressing through the DDC (Table 2.1). These instructions are intended for hybrid-capability code (see Section 2.6.4, [7, p151]), so are more limited and don't support immediate offsets. The behaviour of basic RISC-V load/store opcodes changes to either use capabilities as memory references or use integer addressing via the DDC, depending on the encoding mode. If any instruction tries to dereference an invalid capability, it raises a synchronous exception.

2.6.3 Capability and Integer encoding mode

CHERI-RISC-V specifies two encoding modes, selected using a flag in the PCC `flags` field. *Capability mode* modifies the behaviour of pre-existing instructions to take address operands as capabilities. This makes the basic load/store instruction behaviour exactly equivalent to newly introduced counterparts: e.g. `L[BWHD][U] == L[BWHD][U].CAP`. The DDC may still be used in this mode via the new instructions e.g. `S[BWHD].DDC`.

Integer mode seeks to emulate a standard CHERI-less RISC-V architecture as much as possible. All pre-existing RISC-V memory access instructions take address operands as integers, which are dereferenced relative to the DDC[9]. This makes the basic load/store instruction behaviour exactly equivalent to newly introduced counterparts: e.g. `L[BWHD][U] == L[BWHD][U].DDC`. The new instructions may still be used to dereference and inspect capability registers, but all other instructions access registers in an integer context i.e. ignoring the upper bits and tag from merged register files.

2.6.4 Pure-capability and Hybrid compilation modes

CHERI-Clang[10], the main CHERI-enabled compiler, supports two ways to compile CHERI-RISC-V which map to the different encoding modes.

Pure-capability mode treats all pointers as capabilities, and emits pre-existing RISC-V instructions that expect to be run in capability mode[11].

Hybrid mode treats pointers as integer addresses, dereferenced relative to the DDC, unless they are annotated with `__capability`. This mode emits pre-existing RISC-V instructions that take integer operands, and uses capabilities through the new instructions. All capabilities in hybrid mode are created manually by the program by copying and shrinking the DDC.

Hybrid mode allows programs to be gradually ported to CHERI, making it very easy to adopt on legacy/large codebases. Any extensions to the model (e.g. CHERI-RVV) should try and retain this property.

2.6.5 Capability relocations

Summarizes [8, Section 4.4, Appendix C]

Binary applications compiled in pure-capability mode require some "global" capabilities to exist at startup, e.g. the capability which points to the `main()` function. It would be a security risk to synthesize these capabilities from thin air, or to allow the binary file itself to contain tag bits.

Instead, CHERI ELF binaries contain a set of requested "relocations" (the `__cap_relocs` section) which instruct the runtime environment to create capabilities with specific permissions and bounds in specific places. This process uses the normal CHERI capability instructions, so any invalid requests will cause a program crash, maintaining security. Further complexity is introduced with dynamic linking, and in the future these relocations may change format, both described in TR-949[8], but the above description is sufficient to understand the rest of this paper.

[9] Of course, the DDC must be valid when it is used in this mode, and all bounds checks etc. must still pass.
[10] https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html
[11] This wasn't derived from documentation, but instead from manual inspection of emitted code.

CHAPTER 3

Hardware emulation investigation

In order to experiment with integrating CHERI and RVV, we implemented a RISC-V emulator in the Rust programming language named `riscv-v-lite`. The emulator can partially emulate four unprivileged[1] RISC-V ISAs (Table 3.1), and was also used as the base for capabilities-in-vectors research (Chapter 5). This chapter explores the development of the emulator, the implementation of CHERI support (including supplementary libraries), the addition of vector support, and the conclusions drawn about CHERI-RVV.

	Architecture	Extensions
32-bit	`rv32imv`	Multiply, CSR, Vector[1]
64-bit	`rv64imv`	Multiply, CSR, Vector[1]
64-bit	`rv64imvxcheri`	Multiply, CSR, Vector[1], CHERI
64-bit	`rv64imvxcheri-int`	Multiply, CSR, Vector[1], CHERI (Integer)

[1] Floating-point parts of the vector extension are not supported.

Table 3.1: `riscv-v-lite` supported architectures

3.1 Developing the emulator

Each architecture is simulated in the same way. A `Processor` struct holds the register file and memory, and a separate `ProcessorModules` struct holds the ISA modules the architecture can use. Each ISA module uses a "connector" struct to manipulate data in the `Processor`. For example, the RV64 Integer ISA's connector contains the current PC, a virtual reference to a register file, and a virtual reference to memory. This allows different `Processor` structs (e.g. a normal RV64 and a CHERI-enabled RV64) to reuse the same ISA modules despite using different register file implementations.

[1] i.e. entirely bare-metal without privilege levels for OSs or hypervisors.

Each `Processor` implements a single stage pipeline. Instructions are fetched, decoded with a common decoder function[2], and executed. The processor asks each ISA module in turn if it wants to handle the instruction, and uses the first module to say yes. If the ISA module returns a new PC value it is immediately applied, otherwise it is automatically incremented. This structure easily represents basic RISC-V architectures, and can scale up to support many different new modules.

3.1.1 Emulating CHERI

Manipulating CHERI capabilities securely and correctly is a must for any CHERI-enabled emulator. Capability encoding logic is not trivial by any means, so the `cheri-compressed-cap` C library was re-used rather than implementing it from scratch. Rust has generally decent interoperability with C, but some of the particulars of this library caused issues.

3.1.1.1 `rust-cheri-compressed-cap`

`cheri-compressed-cap` provides two versions of the library by default, for 64-bit and 128-bit capabilities, which are generated from a common source through extensive use of the preprocessor. Each variant defines a set of preprocessor macros (e.g. the widths of various fields) before including two common header files `cheri_compressed_cap_macros.h` and `cheri_compressed_cap_common.h`. The latter then defines every relevant structure or function based on those preprocessor macros. For example, a function `compute_base_top` is generated twice, once as `cc64_decompress_mem` returning `cc64_cap_t` and another time as `cc128_decompress_mem` returning `cc128_cap_t`. Elegantly capturing both sets was the main challenge for the Rust wrapper.

One of Rust's core language elements is the Trait - a set of functions and "associated types" that can be *implemented* for any type. This gives a simple way to define a consistent interface: define a trait `CompressedCapability` with all of the functions from `cheri_compressed_cap_common.h`, and implement it for two empty structures `Cc64` and `Cc128`. In the future, this would allow the Morello versions of capabilities to be added easily. A struct `CcxCap<T>` is also defined which uses specific types for addresses and lengths pulled from a `CompressedCapability`. For example, the 64-bit capability structure holds a 32-bit address, and the 128-bit capability a 64-bit address.

128-bit capabilities can cover a 64-bit address range, and thus can have a length of 2^{64}. Storing this length requires 65-bits, so all math in `cheri_compressed_cap_common.h` uses 128-bit length values. C doesn't have any standardized 128-bit types, but GCC and LLVM provide so-called "extension types" which are used instead. Although the x86-64 ABI does specify how 128-bit values should be stored and passed as arguments[23], these rules do not

[2]The decoder, and therefore all emulated processors, doesn't support RISC-V Compressed instructions.

seem consistently applied[3]. This causes great pain to anyone who needs to pass them across a language boundary.

Rust explicitly warns against passing 128-bit values across FFI, and the Clang User's Manual even states that passing `i128` by value is incompatible with the Microsoft x64 calling convention[4]. This could be resolved through careful examination: for example, on LLVM 128-bit values are passed to functions in two 64-bit registers[5], which could be replicated in Rust by passing two 64-bit values. For convenience, we instead rely on the Rust and Clang compilers using compatible LLVM versions and having identical 128-bit semantics.

The CHERI-RISC-V documentation contains formal specifications of all the new CHERI instructions, expressed in the Sail architecture definition language[6]. These definitions are used in the CHERI-RISC-V formal model[7], and require a few helper functions (see [7, Chapter 8.2]). To make it easier to port the formal definitions directly into the emulator the `rust-cheri-compressed-cap` library also defines those helper functions.

The above work is available online[8], and includes documentation for all C functions (which is not documented in the main repository). That documentation is also available online[9] and partially reproduced in Appendix A.

3.1.1.2 Integrating into the emulator

Integrating capabilities into the emulator was relatively simple thanks to the modular emulator structure. A capability-addressed memory type was created, which wraps a simple integer-addressed memory in logic which performs the relevant capability checks. For integer encoding mode, a further integer-addressed memory type was created where integer addresses are bundled with the DDC before passing through to a capability-addressed memory (see Fig. 3.1). Similarly, a merged capability register file type was created that exposed integer-mode and capability-mode accesses. This layered approach meant code for basic RV64I operations did not need to be modified to handle CHERI at all — simply passing the integer-mode memory and register file would perform all relevant checks.

Integrating capability instructions was also simple. Two new ISA modules were created: `XCheri64` for the new CHERI instructions, and `Rv64imCapabilityMode` to override the behaviour of legacy instructions in capability-encoding-mode (see Fig. 3.2). The actual Processor structure was left mostly unchanged. Integer addresses were changed to capabilities throughout, memory and register file types were changed as described above, and the PCC/DDC were added.

[3] See `https://godbolt.org/z/qj43jssr6` for an example.
[4] `clang/docs/UsersManual.rst:3384` in `llvm/llvm-project` (`release/13.x`) on GitHub
[5] `clang/lib/CodeGen/TargetInfo.cpp:2811` in `llvm/llvm-project` (`release/13.x`) on GitHub
[6] `rems-project/sail` on Github
[7] `CTSRD-CHERI/sail-cheri-riscv` on Github
[8] **(redacted for anonymity)**
[9] **(redacted for anonymity)**

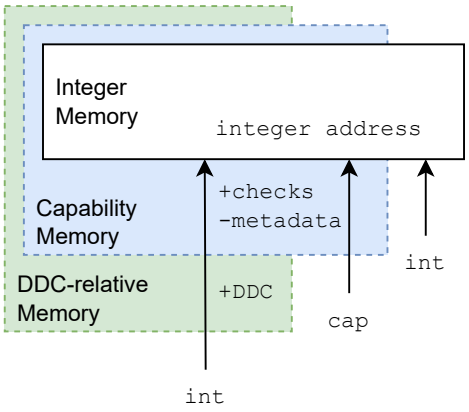


Figure 3.1: Emulator memory structure

if new CHERI instruction **then**
    handle with `XCheri64`
else if basic `rv64` instruction **then**
    if in capability encoding mode **then**
        handle with `Rv64imCapabilityMode`
    else
        wrap memory with DDC-relative
        handle with `Rv64im`
    end if
else if vector instruction **then**
    if in capability encoding mode **then**
        handle with vector unit
    else
        wrap memory in DDC-relative
        handle with vector unit
    end if
end if

Figure 3.2: Example algorithm for emulating `rv64imvxcheri`

The capability model presented by the C/Rust library has one flaw. Each `CcxCap` instance stores capability metadata (e.g. the uncompressed bounds) as well as the compressed encoding. This makes it potentially error-prone to represent untagged integer data with `CcxCap`, as the compressed and uncompressed data may not be kept in sync and cause inconsistencies later down the line. `CcxCap` also provides a simple interface to set the tag bit, without checking whether that is valid. The emulator introduced the `SafeTaggedCap` to resolve this: a sum type which represents either a `CcxCap` with the tag bit set, or raw data with the tag bit unset. This adds type safety, as the Rust compiler forces every usage of `SafeTaggedCap` to consider both options, preventing raw data from being interpreted as a capability by accident and enforcing Provenance.

The final hurdle was the capability relocations outlined in Section 2.6.5. Because we're emulating a bare-metal platform, there is no operating system to do this step for us. A bare-metal C function has been written to perform the relocations[10], which could be compiled into the emulated program. We decided it would be quicker to implement this in the simulator, but in the future we should be able to perform the relocations entirely in bare-metal C.

3.1.2 Emulating vectors

Vector instructions are executed by a Vector ISA module, which stores all registers and other state. `VLEN` is hardcoded as 128-bits, chosen because it's the largest integer primitive provided by Rust that's large enough to hold a capability. `ELEN` is also 128-bits, which isn't supported by the specification, but is required for capabilities-in-vectors (Chapter 5). Scaling `VLEN` and

[10] `src/crt_init_globals.c` in `CTSRD-CHERI/device-model` on GitHub

31 29	28	27 26	25	24 20	19 15	14 12	11 7	6 0
nf	mew	mop	vm	umop/rs2/vs2	rs1	width	vd	opcode

Table 3.2: Relevant parameters of floating-point and vector load/store encoding

ELEN any higher would require the creation and integration of new types that were more than 128-bits long.

To support both CHERI and non-CHERI execution pointers are separated into an address and a *provenance*[11]. The vector unit retrieves an address + provenance pair from the base register, generates a stream of addresses to access, then rejoins each address with the provenance to access memory. When using capabilities, provenance is defined in terms of the base register e.g. "the provenance is provided by capability register X", or defined by the DDC in integer mode[12]. On non-CHERI platforms the vector unit doesn't check provenance.

Arithmetic and configuration instructions are generally simple to implement, so aren't covered here. The emulator splits vector memory accesses into three phases: decoding, checking, and execution. A separate decoding stage may technically not be necessary in hardware (especially the parts checking for errors and reserved instruction encodings, which a hardware platform could simply assume won't happen), but it allows each memory access instruction to be classified into one of the five archetypes outlined in Section 2.5. It is then easy to define the checking and execution phases separately for each archetype, as the hardware would need to do.

3.1.2.1 Decoding phase

Decoding is split into two steps: finding the encoded **nf** and element widths, then interpreting them based on the encoded archetype. Vector memory accesses reuse instruction encodings from the F extension's floating-point load/store instructions, which encode an "element width" in the **mew** and **width** bits (see Table 3.2). The vector extension adds four extra width values which imply the access is vectorized (see Table E.2). If any of these values are found, the instruction is interpreted as a vector access and **nf** is extracted.

Once the generic parameters are extracted, the addressing method is determined from **mop** (Unit, Strided, Indexed-Ordered, or Indexed-Unordered). If a unit access is selected, the second argument field **umop** selects a unit-stride archetype (normal access, fault-only-first, whole register, or bytemask). Extra archetype-specific calculations are performed (e.g. computing `EVL = ceil(vl/8)` for bytemask accesses), and the relevant information is returned as a `DecodedMemOp` enum.

[11] The "original allocation the pointer is derived from"[24], or in CHERI terms the bounds within which the pointer is valid.

[12] See Section 3.1.2.4 for the reasoning behind this decision.

3.1.2.2 Fast-path checking phase

The initial motivation for this project was investigating the impact of capability checks on performance. Rather than check each element's access individually, we determine a set of "fast-path" checks which count as checks for multiple elements at once. In the emulator, this is done by computing the "tight bounds" for each access, i.e. the exact range of bytes that will be accessed, and doing a single capability check with that bounds. Section 3.2 describes methods for calculating the "tight bounds" for each access type, and ways that architectural complexity can be traded off to calculate *wider* bounds.

If the tight bounds don't pass the capability check, the emulator raises an imprecise trap and stops immediately. In the case of fault-only-first loads, where synchronous exceptions (e.g. capability checks) are explicitly handled, the access continues regardless and elements are checked individually. This is also the expected behaviour if a capability check for *wider* bounds fails. The emulator deviates from the spec in that `vstart` is *not* set when the tight bounds check fails, as it does not know exactly which element would have triggered the exception. As noted in Section 3.2, a fully compliant machine must check each access to find `vstart` in these cases.

3.1.2.3 Execution phase

If the fast-path check deems it appropriate, the emulator continues execution of the instruction in two phases. First, the mapping of vector elements to accessed memory addresses is found. The code for this step is independent of the access direction, and an effective description of how each type of access works. It can be found in Appendix B.4. The previously computed tight bounds are sanity-checked against these accesses, and the accesses are actually performed.

3.1.2.4 Integer vs. Capability encoding mode

As noted in Section 2.6.3, CHERI-RISC-V defines two execution modes that the program can switch between. In Integer mode "address operands to existing RISC-V load and store opcodes contain integer addresses" which are implicitly dereferenced relative to the default data capability, and in Capability mode those opcodes are modified to use capability operands.

Integer mode was included in the interests of maintaining compatibility with legacy code that hasn't been adapted to capabilities. As similar vector code may also exist, CHERI-RVV treats vector memory access instructions as "existing RISC-V load and store opcodes" and requires that they respect integer/capability mode.

We do not define new mode-agnostic instructions, like `S[BHWD][U].CAP` and `S[BHWD].DDC` (Section 2.6.2), which means vector programs cannot mix capability and integer addressing without changing encoding modes. This may make incremental adoption more difficult, and in the future we should examine existing vanilla RVV programs to determine if it's worth adding

those instructions.

3.2 Fast-path calculations

A fast-path check can be performed over various sets of elements. The emulator chooses to perform a single fast-path check for each vector access, calculating the tight bounds before starting the actual access, but in hardware this may introduce prohibitive latency. This section describes the general principles surrounding fast-paths for CHERI-RVV, notes the areas where whole-access fast-paths are difficult to calculate, and describes possible approaches for hardware.

3.2.1 Possible fast-path outcomes

In some cases, a failed address range check may not mean the access fails. The obvious case is fault-only-first loads, where capability exceptions may be handled without triggering a trap. Implementations may also choose to calculate wider bounds than accessed for the sake of simplicity, or even forego a fast-path check altogether. Thus, a fast-path check can have four outcomes depending on the circumstances.

A Success means no per-access capability checks are required. Likely-Failure and Unchecked results mean each access must be checked, to see if any of them actually raise an exception. Unfortunately, accesses still need to be checked under Failure, because both precise and imprecise traps need to report the offending element in `vstart`[13].

Because all archetypes may have Failure or Likely-Failure outcomes, hardware must provide a fallback slow-path for each archetype which checks/performs each access in turn. In theory, a CHERI-RVV specification could relax the `vstart` requirement for imprecise traps, and state that all capability exceptions trigger imprecise traps. In this case, only archetypes that produce Likely-Failure outcomes need the slow-path. However, it is likely that for complexity reasons all masked accesses will use wide ranges, thus producing Likely-Failure outcomes and requiring slow-paths for all archetypes anyway. Because the Likely-Failure and Failure cases require the slow-path anyway, computing the fast-path can only be worthwhile if Success is the common case.

3.2.2 Whole-access fast-paths

It is technically possible to calculate a fast-path for the entirety of an access (see Appendix C), but for some situations it may be equally/more expensive than checking each access. For

[13] In very particular cases, e.g. unmasked unit-strided accesses where `nf = 1`, the capability bounds could be used to calculate what the offending element must have been. We believe this is too niche of a use case to investigate further, particularly given the complexity of the resulting hardware.

Success	All accesses will succeed
Failure	At least one access *will* raise an exception
Likely-Failure *or* Unchecked	At least one access *may* raise an exception

(a) Possible fast-path outcomes

if range is within capability **then**
    Success
else if range is wide **then**
    Likely-Failure
else if fault-only-first **then**
    Likely-Failure
else
    Failure
end if

(b) Algorithm

Figure 3.3: Fast-path outcomes

example, the bounds for masked accesses depend on finding the minimum and maximum active indices, which in hardware may require a linear scan. Indexed accesses require finding the minimum/maximum offset values, which likely requires an expensive parallel reduction over all/some elements. In these cases hardware implementations could defer to the slow-path on all masked/indexed accesses, or for masked accesses use the wider, unmasked bounds and generate Likely-Failure outcomes. Unit and strided accesses are much easier to handle.

Arbitrarily strided accesses (which may have positive, negative, or zero-valued strides) are relatively simple to calculate. After calculating the segment width (i.e. number of fields $*$ element width) the full bounds just depends on the sign of the stride (Eq. (3.2.1)). Unit-stride accesses simplify this further, because the stride is equal to the segment width and guaranteed to be positive (Eq. (3.2.2)).

$$\texttt{base} + \begin{cases} \texttt{[vstart * stride, (evl - 1) * stride + nf * eew)} & \texttt{stride} \geq 0 \\ \texttt{[(evl - 1) * stride, vstart * stride + nf * eew)} & \texttt{stride} < 0 \end{cases} \tag{3.2.1}$$

Tight bounds for strided access

$$\texttt{base} + \texttt{[vstart * nf * eew, evl * nf * eew)} \tag{3.2.2}$$

Tight bounds for unit-stride access

Ultimately, the potential up-front latency seemed like a dealbreaker for this approach. We turned our attention to fast-pathing smaller groups of elements.

3.2.3 m-element known-range fast-paths

A hardware implementation of a vector unit may be able to issue m requests within a set range in parallel. For example, elements in the same cache line may be accessible all at once. In these cases, checking elements individually would either require m parallel bounds checks, m

checks' worth of latency, or something inbetween. In this subsection we consider a fast-path check for m elements.

Capability checks can be split into two steps: address-agnostic (e.g. permissions checks, bounds decoding) and address-dependent (e.g. bounds checks). Address-agnostic steps can be performed before any bounds checking, and should add minimal start-up latency (bounds decoding must complete before the checks anyway, and permission checks can be performed in parallel). Once the bounds are decoded the actual checks consist of minimal logic[14], so a fast-path must have very minimal logic to compete.

We first consider unit and strided accesses, and note two approaches. First, one could amortize the checking logic cost over multiple sets of m elements by operating in terms of cache lines. Iterating through all accessed cache lines, and then iterating over the elements inside, allows the fast-path to hardcode the bounds width and do one check for multiple cycles of work (if cache lines contain more than m elements). Cache-line-aligned allocations benefit here, as all fast-path checks will be in-bounds i.e. Successful, but misaligned data is guaranteed to create at least one Likely-Failure outcome per access (requiring a slow-path check). Calculating tight bounds for the m accessed elements per cycle could address this.

For unit and strided accesses, the bounds occupied by m elements is straightforward to calculate, as the addresses can be generated in order. The minimum and maximum can then be picked easily to generate tight bounds. An m-way multiplexer is still required for taking the minimum and maximum, because `evl` and `vstart` may not be m-aligned. If m is small, this also neatly extends to handle masked/inactive elements. This may use less logic overall than m parallel bounds checks, depending on the hardware platform[15], but it definitely uses more logic than the cache-line approach. Clearly, there's a trade-off to be made.

Indexed fast-paths are more complicated, because the addresses are unsorted. The two approaches above have different advantages for indexed accesses. If the offsets/indices are spatially close, just not sorted, cache line checks may efficiently cover all elements. An implementation could potentially cache the results, and refer back for each access, instead of trying to iterate through cache lines in order. Otherwise a m-way parallel reduction could be performed to find the min and max, but that would likely take up more logic than m comparisons. This may be a moot point depending on the cache implementation though - if the m accesses per cycle must be in the same cache line, and the addresses are spread out, you're limited to one access and therefore one check per cycle regardless.

In summary, there are fast-path checks that consume less logic than m parallel checks in certain circumstances. Even though a slow-path is always necessary, it can be implemented in a slow way (e.g. doing one check per cycle) to save on logic. Particularly if other parts of the system rely on constraining the addresses accessed in each cycle, a fast-path check can take

[14] Likely requires two arithmetic operations per element, for checking against the top and bottom bounds.

[15] e.g. on FPGAs multiplexers can be relatively cheap.

advantage of those constraints.

3.3 Going beyond the emulator

The emulator is a single example of a conformant CHERI-RVV implementation, and does not exercise every part of the specification. Four properties stand out:

- The emulator assumes all element accesses are naturally aligned, but the spec allows misaligned accesses.
- The emulator doesn't consider multiple hardware threads, essentially assuming all accesses are atomic.
- Segments/elements are always accessed in order, despite the spec not enforcing ordering
- Imprecise traps are used for all exceptions - precise trap behaviour is not explored.

This section notes how relaxed access ordering and precise exceptions may affect the hardware in ways not previously explored.

3.3.1 Misaligned accesses

Implementations are allowed to handle vector accesses that are not aligned to the size of the element. This support is independent of misaligned scalar access support, so if e.g. misaligned 64-bit scalar accesses are allowed, misaligned vector accesses of 64-bit elements do *not* have to be allowed.

Changing the emulator to allow misaligned accesses of integer data would not have any impact on CHERI correctness. Capability loads/stores must be aligned to `CLEN`[7, Section 3.5.2], and an implementation cannot change this. Writing misaligned integer values across a `CLEN` boundary would need to make sure to zero the tag bit on both regions, but this applies to scalar implementations as much as vector ones. Alignment only impacts CHERI-RVV to the extent that it impacts capabilities-in-vectors (Section 5.2).

3.3.2 Atomicity of accesses/General memory model

Vector memory instructions are specified to follow the RISC-V Weak Memory Ordering model[2][16], although this model hasn't been fully explained in terms of vectors yet. RVWMO defines a global order of "memory operations": atomic operations that are either loads, stores, or both[6, Chapter 14]. The RVWMO spec assumes all memory instructions create exactly one

[16] Behaviour under the Total Store Ordering extension hasn't been defined.

memory operation but calls out that once the vector model is formalized, vector accesses may be defined to create multiple operations.

The RVV spec states "vector misaligned memory accesses follow the same rules for atomicity as scalar misaligned memory accesses", i.e. that misaligned accesses may be decomposed into multiple memory operations of any granularity[17]. This is the only mention of atomicity in that document.

Again, atomicity of integer data doesn't really impact the fusion of CHERI and RVV, as long as tag bits are correctly zeroed on all integer writes. However, it does impact capabilities-in-vectors (Section 5.2).

3.3.3 Relaxed access ordering and precise traps

Ordering is only enforced insofar as it is observable. The only instructions that are forced to perform their accesses in order are indexed-ordered accesses, which can be used to write to e.g. I/O regions where order matters, and instructions that trigger precise traps. Precise traps require `vstart` to be set to a value such that all elements before `vstart` have completed their accesses, and all accesses on/after `vstart` have not completed or are idempotent.

If a vector memory access instruction is 1. not indexed-ordered and 2. guaranteed not to trigger a precise trap[18] then it may execute out of order. This does not affect CHERI-RVV in any way.

3.4 Testing and evaluation

We tested the emulator using a set of test programs described in Sections 4.3 and 5.2, and found that all instructions were implemented correctly.

[17] e.g. each byte could be written in a separate access.

[18] Even instructions that *would* trigger precise traps but are guaranteed not to throw an exception or respond to asynchronous interrupt may execute out of order.

Hypothesis H-1 - Feasibility

> *It is possible to use CHERI capabilities as memory references in all vector instructions.*

This is true. All vector memory access instructions index the scalar general-purpose register file to read the base address, and CHERI-RVV implementations can simply use this index for the scalar capability register file instead. This can be considered through the lens of adding CHERI to any RISC-V processor, and in particular adding Capability mode to adjust the behaviour of legacy instructions. RVV instructions can have their behaviour adjusted in exactly the same way as the scalar memory access instructions.

That approach then scales to other base architectures that have CHERI variants. For example, Morello's scalar Arm instructions were modified to use CHERI capabilities as memory references[25, Section 1.3], so one may simply try to apply those modifications to e.g. Arm SVE instructions. This only works where Arm SVE accesses memory references in the same way as scalar Arm instructions did i.e. through a scalar register file.

Arm SVE has some addressing modes like `u64base`, which uses a vector as a set of 64-bit integer addresses[26]. This has more complications, because simply dereferencing integer addresses without a capability is insecure. Would a CHERI version convert this mode to use capabilities-in-vectors, breaking compatibility with legacy code that expects integer references? Another option would be to only enable this instruction in Integer mode, and dereference relative to the DDC. It's possible to port this to CHERI, but requires further investigation and thought.

Hypothesis H-2 - Fast-path checks

> *The capability bounds checks for vector elements within a known range (e.g. a cache line) can be performed in a single check, amortizing the cost.*

This is also true, at least for Successful accesses. Because the RVV spec requires that the faulting element is *always* recorded[2, Section 17], a Failure due to a capability violation requires elements to be checked individually. CHERI-RVV could change the specification so the faulting element doesn't need to be calculated, which would make Failures faster, but that still requires Likely-Failures to take the slow-path.

There are many ways to combine the checks for a set of vector elements, which can take advantage of the range constraints. For example, a unit-stride access could a hierarchy of checks: cache-line checks until a Likely-Failure, then tight m-element bounds until a Likely-Failure, then the slow-path. However, the choice of fast-path checks is inherently a trade-off between latency, area, energy usage, and more. Picking the right one for the job is highly dependent on the existing implementation, and indeed an implementation may decide that parallel per-element checks is better than a fast-path.

CHAPTER 4

The CHERI-RVV software stack

This chapter explores the current state of the CHERI-RVV software stack: mainstream compiler support for vanilla RVV (Section 4.1) and the modifications required to bring support to CHERI-Clang (Section 4.2). The software hypotheses are tested with this knowledge (Section 4.3), and we recommend a set of changes to bring CHERI-Clang support to par with other compilers (Section 4.4).

4.1 Compiling vector code

Modern compilers provide many ways to generate vectorized code. While this support is very advanced for well established vector models, like x86-64 AVX, newer vector models like RVV don't have as many options. It can even be difficult to get the compiler to generate any vector instructions at all. This section examines support across the Clang and GCC compilers for various vectorization methods on RVV.

4.1.1 Available compilers

Compiler support for RVV varies. On Clang 13 and other LLVM-13-based compilers, version 0.1(?[1]) of the vector specification is supported as an experimental extension. Clang/LLVM 14 and up support RVV v1.0.

GCC is an interesting case — there is a version based on RISC-V GCC 10.1 that partially supports RVV (see Appendix D.3), but it was left untouched for a year and deleted as of 17th May 2022. GCC RVV support has also been deprioritized in favour of LLVM[2]. See Appendix D.5 for more information on finding and building this version, and Table D.1 for the required command-line arguments to enable RVV.

[1] It is difficult to verify the actual corresponding version, because there is no readily available specification for v0.1, and the extension supports instructions only present from v0.8 such as whole register accesses.

[2] https://github.com/riscv-collab/riscv-gcc/issues/320

4.1.2 Automatic vectorization

Compilers with auto-vectorization can automatically create vectorized code from a scalar program. For example, a scalar loop over an array that increments each element could be converted to a vectorized loop that increments multiple elements at once. Clang and GCC support auto-vectorization in Arm SVE, explored further in Section 4.1.5, but don't yet support it for RVV. Arm SVE and RVV are quite similar, so there shouldn't be anything blocking auto-vectorization for RVV, it just requires engineering effort.

4.1.3 Vector intrinsics

"Intrinsics" are functions defined by the compiler that can invoke low-level functionality and instructions directly for a specific architecture. When automatic vectorization is not available, intrinsics are the next best thing — they aren't portable across different ISAs, but present a familiar high-level interface (function calls) that gives fine-grained control over instructions. The compiler then handles low-level decisions like register allocation under the hood, and sometimes may provide extra functionality for ease of use.

RVV has a comprehensive set of vector intrinsics[27]. With these, the general strip-mining loop is easy to construct:

1. Use a `vsetvl` intrinsic to get the vector length for this iteration.
2. Allocate vector registers by declaring variables with vector types (e.g. `vuint32m8_t` represents 8 registers worth of 32-bit unsigned integers).
3. Pass the vector length to the computation/memory intrinsics, which operate on the vector variables.

Appendix B.1 contains an example.

4.1.4 Inline assembly

If a compiler doesn't supply complete intrinsics, or if the programmer desires even more control, inline assembly may be used. The programmer gives a string of handwritten assembly code to the compiler, which is parsed and directly inserted into the output code. The compiler still has to understand the instruction, but it doesn't need intrinsics to be present (or functional[3]).

Inline assembly can interact with C code and variables through a template syntax. The programmer inserts a placeholder in the assembly code with a corresponding expression, noting how the expression is stored using a "constraint". For our purposes, constraints enforce that a value is either in a register or in memory.

[3]As described later, CHERI-Clang crashes when intrinsics are used, so we use inline assembly instead.

Using the constraint, the compiler determines how the expression's value is stored, and inserts a reference to it in the assembly string. Because this is done before the assembly string is parsed, and isn't immediately type-checked against the assembly instruction, it can lead to some difficult errors.

Clang and GCC support inline assembly for RVV quite well, and even allows the intrinsic vector types to be referenced by assembly templates (thus making the compiler do register allocation instead of the programmer). The only caveat is that *memory* constraints are not supported by RVV memory accesses. None of the vector memory access instructions support address offsets, unlike their scalar counterparts. Clang always treats the memory constraint as an offset access, even when that offset is zero, so it adds an offset to the assembly string (Fig. 4.1c) making it invalid. To get around this, one must use the pointer itself with a *register* constraint (Fig. 4.1d).

```
int* ptr; int val; vint32m1_t vec_val;
```

(a) Preamble

```
// "ld a0, 0(a0)" - valid
asm ("ld %0, %1"
        : "=r"(val)
        : "m"(*ptr));
```

(b) Load scalar from memory

```
// "vle32.v v8, 0(a0)" - invalid
asm ("vle32.v %0, %1"
        : "=vr"(vec_val)
        : "m"(*ptr));
```

(c) Failed attempt to load vector from memory

```
// "vle32.v v8, (a0)" - valid
asm ("vle32.v %0, (%1)"
        : "=vr"(vec_val)
        : "r"(ptr));
```

(d) Load vector from pointer in register

Figure 4.1: Inline assembly examples
https://godbolt.org/z/rW9orr66a

Broadly speaking, inline assembly supports more RVV instructions than intrinsics do. It is used extensively in the testbench code for the evaluation (Section 4.3) alongside intrinsics where possible.

4.1.5 RVV vs. Arm SVE

Arm SVE uses a similar model to RVV, where the vector length may scale between 128 and 2048[4] and the instructions are designed to be totally agnostic across different platforms[1]. Arm have released a C language extension to support SVE development ([28]), supported by the Arm Compiler for Embedded[5], Clang, and GCC. They support all of the previously examined vectorization types.

Auto-vectorization is supported, and the main focus of the user guide ([26]) is helping the compiler decide whether to auto-vectorize. Intrinsics are also supported, and seem to cover all of the SVE instructions, but take a slightly different approach to RVV. Arm SVE intrinsics do not directly map to available instructions, but aim to "provide a regular interface and leave the compiler to pick the best mapping to SVE instructions", while RVV intrinsics (at least for memory) tend to map 1:1 to existing instructions. Arm's approach gives more flexibility for future extensions, as the same intrinsics could be compiled to new instructions with newer compilers.

Arm SVE also supports inline assembly, but the experience is notably worse than for RVV. The two standout issues are a lack of register allocation and the use of condition code flags for branching. Unlike RVV, the intrinsic types for vector values cannot be referenced in inline assembly[1], so all vector registers must be allocated and tracked by the programmer. Arm SVE's equivalent of `vsetvl`, the `while` family[28], do not return the number of updated elements, and instead set the condition flags based on how many elements are updated. Because there is no way to branch based on the condition flags in C, the programmer must manually insert a label for the top of the loop, and a branch to that label, which is more error prone than the RVV method. See Appendix B.3 for examples of Arm SVE code with auto-vectorization, intrinsics, and inline ASM.

[4] RVV slightly differs here, as it allows VLEN smaller than 128.

[5] https://developer.arm.com/Tools%20and%20Software/Arm%20Compiler%20for%20Embedded

```
// "vle32.v v8, (ca0)" - valid
asm ("vle32.v %0, (%1)"
        : "=vr"(vec_val)
        : "C"(ptr));
```

(a) Load vector from capability in register

```
#if __has_feature(pure_capabilities)
#define PTR_REG "C"
#else
#define PTR_REG "r"
#endif

// Produces "vle32.v v8, (ca0)"
// or "vle32.v v8, (a0)"
asm ("vle32.v %0, (%1)"
        : "=vr"(vec_val)
        : PTR_REG(ptr));
```

(b) Portable code for CHERI and non-CHERI[1].

[1] This relies on the `pure_capabilities` feature flag, which was added to CHERI-Clang for this project.

Figure 4.2: Inline assembly examples (CHERI)

4.2 Compiling vector code with CHERI-Clang

Current CHERI compiler work is done on CHERI-Clang, a fork of Clang and other LLVM tools that supports capabilities. It's based on LLVM 13, and supports vanilla RVV v0.1, but the vector-related code had not been updated to handle capabilities. This section outlines the changes required to compile vector programs for CHERI-RVV using CHERI-Clang. The required command-line options for CHERI-Clang are noted in Appendix D.2.

4.2.1 Adapting vector assembly instructions to CHERI

LLVM uses a domain-specific language to describe the instructions it can emit for a given target. The RISC-V target describes multiple register sets that RISC-V instructions can use. Vanilla RVV vector memory accesses use the General Purpose Registers (GPR) to store the base address of each access. CHERI-Clang added a GPCR set, i.e. the General Purpose Capability Registers, which use a different register constraint. We created two mutually exclusive versions of each vector access instruction: one for integer mode using a GPR base address; and one for capability mode using GPCR.

With the above changes, inline assembly could be used to insert capability-enabled vector instructions (Fig. 4.2a). However, as this requires using a capability register constraint for the base address, inline assembly code written for CHERI-RVV is not inherently compatible with vanilla RVV. For un-annotated pointers (e.g. `int*`), which are always capabilities in pure-capability code and integers in legacy or hybrid code, a conditional macro can be used to insert the correct constraint (Fig. 4.2b). However, this falls apart in hybrid code for manually annotated pointers (e.g. `int* __capability`) because the macro cannot detect the annotation.

4.2.2 Adapting vector intrinsics to CHERI

Vector intrinsics are another story entirely. When compiling for pure-capability libraries, all attempts to use vector intrinsics crash CHERI-Clang. This is due to a similar issue to inline assembly: the intrinsics (both the Clang intrinsic functions and the underlying LLVM IR intrinsics) were designed to take regular pointers and cannot handle it when capabilities are used instead. Unfortunately the code for generating the intrinsics is spread across many files, and there's no simple way to change the pointers to capabilities (much less changing it on-the-fly for capability vs. integer mode).

It seems that significant engineering work is required to bring vector intrinsics up to scratch on CHERI-Clang. We did experiment with creating replacement wrapper functions, where each function tried to mimic an intrinsic using inline asssembly. These were rejected for two reasons: the overhead of a function call for every vector instruction[6], and lack of support for passing vector types as arguments or return values. The RISC-V ABI treats all vector registers as temporary and explicitly states that "vector registers are not used for passing arguments or return values"[29]. CHERI-Clang would try to return them by saving them to the stack, but this had its own issues.

4.2.3 Storing scalable vectors on the stack

If a program uses more data than can fit in registers, or calls a function which may overwrite important register values, the compiler will save those register values to memory on the stack. Because vector registers are temporary, and thus may be overwritten by called functions, they must also be saved/restored from the stack (see Fig. B.1). This also applies to multiprocessing systems where a process can be paused, have the state saved, and resume later. RVV provides the whole-register memory access instructions explicitly to make this process easy[2, Section 7.9].

CHERI-Clang contains an LLVM IR pass[7] which enforces strict bounds on so-called "stack capabilities" (capabilities pointing to stack-allocated data), which by definition requires knowing the size of the data ahead of time. This pass assumes all stack-allocated data has a static size, and crashes when dynamically-sized types e.g. scalable vectors are allocated. It is therefore impossible (for now) to save vectors on the stack in CHERI-Clang, although it's clear that it's theoretically possible. For example, the length of the required vector allocations could be calculated based on `VLEN` before each stack allocation is performed, or if performance is a concern stack bounds for those allocations could potentially be ignored altogether. These possibilities are investigated further in the next section.

[6] We tried using preprocessor macros instead of real functions, but they are difficult to program and do not support returning values like intrinsics do.

[7] `llvm/lib/CodeGen/CheriBoundAllocas.cpp` in `CTSRD-CHERI/llvm-project` on GitHub

4.3 Testing and evaluation

We developed a self-checking test program for the emulator to execute, which helped gather information for the hypotheses and find bugs in the compiler and emulator. Initially it was hand-written, but in order to test a wide range of `vtypes` we began generating it with a Python script. It consists of fifty-seven tests of different vector memory access archetypes under various configurations (Table 4.1).

The test code uses intrinsics wherever the compiler supports them (see Appendix D.3), and falls back to inline assembly otherwise. Inline assembly uses the preprocessor macro from Fig. 4.2b to handle CHERI and non-CHERI platforms.

The tests are run inside *harnesses*, which provide setup and self-checking code for common cases: The Vanilla harness tests a simple `memcpy` between two arrays; Masked tests that every other element is copied, not all of them; Segmented tests a `memcpy` into four separate output arrays, each a different field of a four-field structure. There is also a special test for fault-only-first: FoF loads are performed at the edge of mapped memory, and the test shows that out-of-bounds exceptions are swallowed and `vl` is reduced accordingly. All tests were successful when they ran, but some testbenches could not be built with some compilers. The full set of test results is available in Appendix F.

Test Scheme	Harness	Compilers
Unit Stride	Vanilla	All
Strided	Vanilla	All
Indexed	Vanilla	All
Whole Register	Vanilla	All
Fault-only-First	Vanilla	All
Unit Stride (Masked)	Masked	All
Bytemask Load	Masked	`llvm-15` only
Unit Stride (Segmented)	Segmented	All
Fault-only-First Boundary	—	All

Table 4.1: `vector_memcpy` test schemes and harnesses

Hypothesis H-3 - Compiling/running legacy code in integer mode

> *Vector code can be compiled in legacy forms (with integer addressing) and function correctly on CHERI with no source code changes.*

This is true for CHERI-RVV, when running the compiled programs in integer mode, as long as the programs only access memory within the DDC.

All vanilla RVV instructions have counterparts with identical encodings and behaviour in CHERI-RVV integer mode, assuming the accessed addresses are all accessible through the DDC.

There are no changes to instruction behaviour that require the compiler's handling of them to change, so a non-CHERI compiler and an integer-mode-CHERI compiler can always produce the same vector instructions from the same code. This does not apply to capability-mode-CHERI, because integer addressing is not supported in capability-mode-CHERI-RVV.

All legacy vector programs should produce equivalent binaries when compiled for integer-mode-CHERI. On top of that, all binaries compiled for vanilla RVV platforms should produce the same results when run on an equivalent integer-mode-CHERI-RVV platform. Both claims assume the program doesn't perform out-of-bounds accesses relative to the DDC.

Hypothesis H-4 - Converting legacy code to pure-capability code

Legacy vector code can be compiled into a pure-capability form with no changes.

This is true for CHERI-RVV, but cannot be done in practice yet. Engineering effort is required to support this in CHERI-Clang. Because this argument concerns source code, all three ways to generate CHERI-RVV instructions must be examined.

Inline Assembly — Unlikely

For GCC-style inline assembly, it is currently impossible for integer-addressed RVV source code to be recompiled in pure-capability mode without modification. Integer-addressed RVV uses general-purpose registers for the base address, but pure-capability instructions require capability registers instead. The base address register can either be specified directly, so must be changed to a capability register; or specified using template syntax and an "r" constraint, which must be changed to a "C" constraint (Figs. 4.1 and 4.2). Using a preprocessor macro (e.g. Fig. 4.2b) could make code portable between non-CHERI and CHERI, but this is still a source code change.

In theory, one could change the behaviour of inline assembly to automatically convert general purpose registers/constraints to capability versions in specific circumstances. However, this can have wide-reaching ramifications, potentially making code more difficult to understand, or even breaking existing code.

Intrinsics — Yes

The current specification for RVV intrinsics uses pointer types for all base addresses[27]. In pure-capability compilers all pointers should be treated as capabilities instead of integers, including those in intrinsics. All RVV memory intrinsics have equivalent RVV instructions, which all use capabilities in pure-capability mode, so changing the intrinsics to match is valid.

Assuming all base address pointers are created in a valid manner (e.g. through `malloc` or monotonic decrease, and not through integer literals), the conversion to pure-capability

should make them all valid capabilities which are compatible with the intrinsics. Therefore well-behaved code using RVV intrinsics should be compilable in pure-capability mode without changes.

This is not currently the case for CHERI-Clang, as RVV memory access intrinsics are broken, but this can be fixed with engineering effort.

Auto-vectorization — Yes

All vanilla RVV instructions have counterparts with identical encodings and behaviour in CHERI-RVV pure-capability mode, assuming the base addresses can be converted to valid capabilities. Any scalar code that can be a) compiled in scalar pure-capability mode[8], and b) auto-vectorized by a legacy RVV compiler, must have an equivalent pure-capability vectorized form. This form could be acquired by performing the auto-vectorization in legacy mode, ensuring all base addresses are available as capabilities, then making the vector instructions use those capabilities. Therefore a pure-capability compiler can always auto-vectorize CHERI-compliant scalar code if some legacy compiler can also auto-vectorize it.

This is not currently possible for CHERI-Clang, as RVV auto-vectorization is not implemented yet. Similar models (e.g. Arm SVE) already have auto-vectorization, so RVV auto-vectorization (and thus CHERI-RVV auto-vectorization) should be possible.

Hypothesis H-5 - Saving vectors on the stack

> *Vector code that saves/restores variable-length vectors to/from the stack can be compiled on CHERI-RVV with no source code changes.*

This is true in theory, but not yet supported by CHERI-Clang in practice. Placing variable-length structures on the stack is possible as long as the length can be known at runtime (and as long as the stack has space, of course). This isn't exclusive to CHERI — to push and pop values on the stack, the stack pointer must be incremented or decremented by the size of the value. Because the length already has to be measured, and CHERI-RISC-V supports setting capability bounds from runtime-computed values, it's entirely possible to correctly set tight bounds for capabilities pointing to variable-length vectors on the stack.

[8] This ensures all memory accesses use valid capabilities.

A minor complication is presented by a note in TR-949[8, Section 3.8.2] concerning "re-materializing bounded stack variables". This section implies LLVM can try to re-create a pointer-to-stack at any time with minimal cost, but this may not be able to apply to vectors. Measuring the bounds requires measuring `VLMAX` by changing `vl`, which could then require saving/restoring the old value. This is only a performance issue, and in the worst case we can just say pointers-to-stack-vectors are not re-materializable, so it isn't a dealbreaker. Further investigation of this issue is left as future work.

Hypothesis H-6 - Running CHERI-RVV code in a multiprocessing system

> *CHERI-vector code can run correctly in multiprocessing systems, where execution may be paused and resumed on interrupts or context switches.*

This requires two conditions: an OS must be able to save and restore vector state, and the vector hardware must support resuming from an interrupted state. The first condition is easy to fulfil by extending the previous hypothesis. If it is possible to save variable-length vectors on the stack, given their length is known at runtime, it must also be possible to save their data on the heap. Some OSs might need to make changes to their "current process state" structure to support variable-length data, and they would also need to allocate space for the `vtype` value, but it is certainly possible.

The second condition can be upheld in two ways. First, if the OS only context switches and services interrupts while the vector hardware is in a complete state (i.e. not partially executing an instruction), then context switches and interrupts are completely transparent to the vector hardware and no changes need to be made. Secondly, if context switches and interrupts can actually interrupt vector instructions partway through, then they can only be cleanly resumed if the vector hardware supports precise traps for the exact instruction being executed.

4.4 Recommended changes for CHERI-Clang

- Build on current work to make all RVV memory access instructions and pseudoinstructions CHERI-compatible.
- Make RVV memory access intrinsics take capabilities as arguments when compiled in pure-capability mode.
- Make the CheriBoundAllocas IR pass handle scalable vectors by finding the length at runtime, and investigate re-materialization of those pointers.
- Bring up CHERI-enabled auto-vectorization in parallel with vanilla auto-vectorization.

CHAPTER 5

Capabilities-in-vectors

Implementing `memcpy` correctly for CHERI systems requires copying the tag bits as well as the data. As it stands, any vectorized `memcpy` compiled and executed on the systems described in Chapters 3 and 4 will not copy the tag bits, because the vector registers cannot store the tag bits and indeed cannot store valid capabilities. `memcpy` is very frequently vectorized, as noted in Section 1.1, so it's vital that CHERI-RVV can implement it correctly. Manipulating capabilities-in-vectors could also accelerate CHERI-specific processes, such as revoking capabilities for freed memory[30].

This chapter examines the changes made to the emulator to support storing capabilities-in-vectors, and determines the conditions required for the related hypotheses to be true. Appendix E.2 lists the changes made and all the relevant properties of the emulator that allow storing capabilities in vectors.

5.1 Changing the emulator

We developed a set of goals based on Hypotheses H-7 to H-9.

- (Hypothesis H-7) Vector registers should be able to hold capabilities
- (Hypothesis H-8) At least one vector memory operation should be able to load/store capabilities from vectors
  - Because `memcpy` should copy both integer and capability data, the vector memory operations should be able to handle both
- (Hypothesis H-9) Vector instructions should be able to affect capabilities in some way
  - Clearing the tag bit on a vector register counts as manipulation

First, we considered the impact on the theoretical vector model. We decided that any operation with elements smaller than `CLEN` cannot output valid capabilities under any circumstances[1], meaning a new element width equal to `CLEN` must be introduced. We set `ELEN = VLEN = CLEN = 128`[2] for our vector unit.

Two new memory access instructions were created to take advantage of this new element width, and the `vsetvl` family were adjusted to support 128-bit values. Similar to the CHERI-RISC-V `LC/SC` instructions, we implemented 128-bit unit-stride vector loads and stores, which took over officially-reserved encodings[3] we expected official versions to use. We have not tested other types of access, but expect them to be noncontroversial. Indexed accesses require specific scrutiny, as they may be expected to use 128-bit offsets on 64-bit systems. The memory instructions had to be added to CHERI-Clang manually, and Clang already has support for setting `SEW=128` in the `vsetvl` family (Table E.1). These instruction changes affected inline assembly only, rather than adding vector intrinsics, because CHERI-Clang only supports inline assembly anyway.

The next step was to add capability support to the vector register file. Our approach to capabilities-in-vectors is similar in concept to the Merged scalar register file for CHERI-RISC-V (Section 2.6.1), in that the same bits of a register can be accessed in two contexts: an integer context, zeroing the tag, or a capability context which maintains the current tag. The only instructions which can access data in a capability context are the aforementioned 128-bit memory accesses[4]. All other instructions will read out untagged integer data and clear tags when writing data.

A new CHERI-specific vector register file was created, where each register is a `SafeTaggedCap` (p36) i.e. either zero-tagged integer data or a valid tagged capability. This makes it much harder to accidentally violate Provenance, and reuses the code path (and related security properties) for accessing capabilities in memory. Just like scalar accesses, vectorized capability accesses are atomic and 128-bit aligned.

5.2 Testing and evaluation

We constructed a second test program to ensure `memcpy` could be performed correctly with capabilities-in-vectors. It copies an array of `Element` structures that hold pointers to static `Base` structures. On CHERI platforms, even in Integer mode, capability pointers are used

[1] This avoids edge cases with masking, where one part of a capability could be modified while the other parts are left alone.

[2] The tag bits are implicitly instead of explicitly included here because `VLEN,ELEN` must be powers of two.

[3] The RVV spec mentions, but does not specify, potential encodings for 128-bit element widths and instructions ([2, p10, p32], Table E.2).

[4] The encoding mode (Section 2.6.3) does not affect register usage: when using the Integer encoding mode, instructions can still access the vector register in a capability context. This is just like how scalar capability registers are still accessible in Integer encoding mode.

and copied. The first test simply copies the data and tests that all the copied pointers still work, which succeeds on all compilers/architectures. The second test is CHERI-exclusive, and invalidates all pointers during the copy process by performing integer arithmetic on the vector registers. The copied pointers are examined to make sure their tag bits are all zeroed, and this test succeeds on both CHERI configurations.

	RV32	RV-64				
	`llvm-13`	`llvm-13`	`llvm-15`	`gcc`	CHERI	CHERI (Int)
Copy	Y	Y	Y	Y	Y	Y
Copy + Invalidate	-	-	-	-	Y	Y

Table 5.1: `vector_memcpy_pointers` results

Hypothesis H-7 - Holding capabilities in vectors

> *It is possible for vector registers to hold capabilities to enable copying without violating CHERI security principles.*

It is possible for a single vector register to hold a capability (and differentiate a capability from integer data) as long as `VLEN = CLEN`. `VLEN` could also be larger, and a compliant implementation must then have `VLEN` be an integer multiple of `CLEN`. In theory, one could also describe a scheme where capabilities must be held by multiple registers together (e.g. `VLEN = CLEN/2` with one tag bit for every two registers), but this would complicate matters.

If an implementation decides, as we did, that elements of width `CLEN` are required to produce capabilities, then $\texttt{VLEN} \geq \texttt{ELEN}$ therefore $\texttt{VLEN} \geq \texttt{CLEN}$. If a short `VLEN` is absolutely essential, one could place precise guarantees on a specific set of instructions to enable it (e.g. `SEW=64, LMUL=2` unit-stride unmasked loads could guarantee atomic capability transfers) but the emulator does not consider this. The CHERI security properties also impose some conditions.

Provenance & Monotonicity

The tag bit must be protected such that capabilities cannot be forged from integer data. The emulator's integer/capability context approach, where the tag bit may only be set on copying a valid capability from memory, and the output tag bit is zeroed on all other accesses, enforces this correctly.

Integrity

Integrity is not affected by how a capability is stored, as long as the other properties are maintained.

Hypothesis H-8 - Sending capabilities between vectors and memory

> *It is possible for vector memory accesses to load and store capabilities from vector registers without violating CHERI security principles.*

For this to be the case, the instructions which can load/store capabilities must fulfil certain alignment and atomicity requirements. They must require all accesses be `CLEN`-aligned, or at least only load valid caps from aligned addresses, because tag bits only apply to `CLEN`-aligned regions. TR-951 states that capability memory accesses must be atomic[7, Section 11.3]. This applies to vectors, even in ways that don't apply to scalar accesses.

Individual element accesses for a vector access must be atomic relative to each other. This is relevant for e.g. a strided store using an unaligned stride, such that one element writes a valid capability and another element overwrites part of that address range. If unaligned 128-bit accesses are allowed, then either the unaligned second element should "win" and clear relevant tag bits, or the first element should "win" and write the full capability atomically. The emulator requires all 128-bit accesses to be aligned so meets this requirement easily.

Provenance

Provenance requires the accesses be atomic as described above, and require that tag bits are copied correctly: the output tag bit must only be set if the input had a valid tag bit. These conditions also apply to scalar accesses.

Monotonicity

These loads/stores do not attempt to manipulate capabilities, so have no relevance to monotonicity.

Integrity

The same conditions for scalar and other vector accesses apply to maintain Integrity: namely that the base capability for each access should be checked to ensure it is valid. The emulator doesn't allow capabilities-in-vectors to be dereferenced directly, but if an implementation allows it those capabilities would also need to be checked.

Hypothesis H-9 - Manipulating capabilities in vectors

> *It is possible for vector instructions to manipulate capabilities in vector registers without violating CHERI security principles.*

The emulator limits all manipulation to clearing the tag bit, achieved by writing data to the register in an integer context. In theory, it's possible to do more complex transformations, which can be proven by implementing each vector manipulation on vector elements as sequential scalar manipulations on scalar elements.

With this method, all pre-existing scalar capability manipulations can become vector manipulations, but the utility seems limited. For example, instructions for creating capabilities or manipulating bounds en masse don't have an obvious use case. If more transformations are added they should be considered carefully, rather than creating vector equivalents for all scalar manipulations. For example, revocation as described in [30] may benefit from a vector equivalent to `CLoadTags`.

Provenance & Monotonicity

Because the only possible manipulations clear the tag bit, it's impossible to create or change capabilities, so Provenance and Monotonicity cannot be violated. Any manipulations that create capabilities, or potentially any manipulations that transfer capabilities from vector registers directly to scalar registers, would require more scrutiny.

Integrity

As stated before, capabilities-in-vectors cannot be dereferenced directly, so there is no impact on Integrity.

CHAPTER 6

Conclusion

This project demonstrated the viability of integrating CHERI with scalable vector models by producing an example CHERI-RVV implementation. This required both research effort in studying the related specifications ([7, 2]), demonstrated in Chapter 2, and a substantial implementation effort demonstrated in Chapters 3 to 5. We produced four software artifacts: a Rust wrapper for the `cheri-compressed-cap` C library (900 lines of code), a RISC-V emulator supporting multiple architecture extensions (5,300 LoC), a fork of CHERI-Clang supporting RVV (400 changed LoC), and test programs for the emulator (3,000 LoC[1]). Developing these artifacts provided enough information to make conclusions for the initial hypotheses (Table 1.1).

6.1 Evaluating hypotheses

Hypothesis H-1 showed that all memory references can be replaced with capabilities in all RVV instructions while maintaining functionality. Hypothesis H-2 then alleviated performance concerns by showing it was possible to combine the required capability checks for all vector accesses, amortizing the overall cost of checking, although with varying practical benefit.

On the software side Hypotheses H-3 and H-4 showed that non-CHERI vectorized code could be run on CHERI systems, and even recompiled for pure-capability platforms with no source code changes, but that CHERI-Clang's current state adds some practical limitations. We developed the `vector_memcpy` test program to show that despite those limitations, it's possible to write correct CHERI-RVV code on current compilers. Hypotheses H-5 and H-6 address the pausing and resuming of vector code, specifically saving and restoring variable-length architectural state, concluding that it is entirely possible but requires software adjustments.

Through a limited investigation of capabilities-in-vectors, Hypotheses H-7 to H-9 showed that a highly constrained implementation could enable a fully-functional vectorized `memcpy`, as demonstrated in the `vector_memcpy_pointers` test program, without violating CHERI

[1] This doesn't include automatically generated code.

security principles. It should be possible to extend the CHERI-RVV ISA with vector equivalents of existing CHERI scalar instructions, but we did not investigate this further.

Overall, it is clear that scalable vector models can be adapted to CHERI without significant loss of functionality. Most of the hypotheses are general enough to cover other scalable models, e.g. Arm SVE, but any differences from RVV's model will require careful examination. Given the importance of vector processing to modern computing, and thus its importance to CHERI, we hope that this research paves the way for future vector-enabled CHERI processors.

6.2 Future work

The stated purpose of this project was to enable future implementations of CHERI-RVV and CHERI Arm SVE. We've shown this is feasible, and we believe our research is enough to create an initial CHERI-RVV specification, but both could benefit from more research on capabilities-in-vectors.

All architectures may benefit from more advanced vectorized capability manipulation. Because these processes are still evolving, it may be wise to standardize the first version of CHERI-RVV based on this dissertation and only add new instructions as required. Once created, the standard can be implemented in CHERI-Clang[2] and added to existing CHERI-RISC-V processors[3].

More theoretically, other vector models could benefit from *dereferencing* capabilities-in-vectors. Arm SVE has addressing modes that directly use vector elements as memory references, as do its predecessors and contemporaries. A draft specification of CHERI-x86 is in the works[7, Chapter 6], and existing x86 vector models like AVX have similar features. This may prove impractical, but this could be mitigated by e.g. replacing these addressing modes with variants of RVV's "indexed" mode. Once this problem is solved, CHERI will be able to match the memory access abilities of any vector ISA it needs to, making it that much easier for industry to adopt CHERI in the long term.

[2] See Section 4.4 for the other required changes to CHERI-Clang.
[3] https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html

References

[1] Nigel Stephens et al. "The ARM Scalable Vector Extension". In: *IEEE Micro* 37.2 (March 2017), pp. 26–39. ISSN: 0272-1732. DOI: 10.1109/MM.2017.35.

[2] *RISC-V "V" Vector Extension*. 20th September 2021. URL: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[3] Robert N M Watson et al. *An Introduction to CHERI*. UCAM-CL-TR-941. September 2019, p. 43.

[4] László Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[5] Robert N.M. Watson et al. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 20–37. ISBN: 978-1-4673-6949-7. DOI: 10/gfpgzz.

[6] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. 13th December 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[7] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. UCAM-CL-TR-951. University of Cambridge, Computer Laboratory, 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.html (visited on 06/10/2021).

[8] Alexander Richardson. *Complete Spatial Safety for C and C++ Using CHERI Capabilities*. UCAM-CL-TR-949. University of Cambridge, Computer Laboratory, June 2020, p. 189. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf.

[9] Andrew Waterman, Krste Asanovic and John Hauser, eds. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. 4th December 2021. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

[10] Matthew Johns and Tom J. Kazmierski. "A Minimal RISC-V Vector Processor for Embedded Systems". In: *2020 Forum for Specification and Design Languages (FDL)*. September 2020, pp. 1–4. DOI: 10/gnrfdb.

[11] Stefano Di Mascio et al. "On-Board Decision Making in Space with Deep Neural Networks and RISC-V Vector Processors". In: *Journal of Aerospace Information Systems* 18.8 (1st August 2021), pp. 553–570. DOI: 10/gnrfch.

[12] *AndesCore NX27V Processor*. Andes Technology. URL: https://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/ (visited on 11/12/2021).

[13] *SiFive Intelligence X280 - SiFive*. sifive.com. URL: https://www.sifive.com/cores/intelligence-x280 (visited on 15/05/2022).

[14] Chen Chen et al. "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-Bit High Performance RISC-V Processor with Vector Extension : Industrial Product". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. May 2020, pp. 52–64. DOI: 10.1109/ISCA45697.2020.00016.

[15] Matheus Cavalcante et al. "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-Nm FD-SOI". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.2 (February 2020), pp. 530–543. ISSN: 1557-9999. DOI: 10/gnrd7v.

[16] Imad Al Assir et al. "Arrow: A RISC-V Vector Accelerator for Machine Learning Inference". 15th July 2021. arXiv: 2107.07169 [cs]. URL: http://arxiv.org/abs/2107.07169 (visited on 11/12/2021).

[17] Kariofyllis Patsidis et al. "RISC-V2: A Scalable RISC-V Vector Processor". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. October 2020, pp. 1–5. DOI: 10/gnfrn3.

[18] Michael Platzer and Peter Puschner. "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation". In: *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Ed. by Björn B. Brandenburg. Vol. 196. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 1:1–1:18. ISBN: 978-3-95977-192-4. DOI: 10/gnfrn2.

[19] Francesco Minervini and Oscar Palomar Perez. "Vitruvius: An Area-Efficient RISC-V Decoupled Vector Accelerator for High Performance Computing" (RISC-V Week - Paris). 4th May 2022. URL: https://www.youtube.com/watch?v=tlC5kMhrh-k (visited on 13/05/2022).

[20] Gopinath Mahale et al. “A RISC-V VPU for Very Long and Sparse Vectors” (RISC-V Week - Paris). March 2021. URL: https://open-src-soc.org/2022-05/media/posters/4th-RISC-V-Meeting-2022-05-03-Gopinath-Mahale-poster.pdf (visited on 13/05/2022).

[21] Jonathan Woodruff et al. “CHERI Concentrate: Practical Compressed Capabilities”. In: (2019), p. 15. DOI: 10/gm9ngf.

[22] Alexandre Joannou et al. “Efficient Tagged Memory”. In: *2017 IEEE International Conference on Computer Design (ICCD)*. November 2017, pp. 641–648. DOI: 10/ghnj26.

[23] H.J. Lu et al., eds. *System V Application Binary Interface v1.0*. 28th January 2018. URL: https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf (visited on 15/05/2022).

[24] Kayvan Memarian et al. “Exploring C Semantics and Pointer Provenance”. In: *Proceedings of the ACM on Programming Languages* 3 (POPL 2nd January 2019), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3290380.

[25] Arm Ltd. *Arm Architecture Reference Manual Supplement - Morello for A-profile Architecture*. 25th June 2021. URL: https://developer.arm.com/documentation/ddi0606/latest.

[26] Arm Ltd. *Arm Compiler Scalable Vector Extension User Guide Version 6.12*. 0612-00. 27th February 2019. URL: https://developer.arm.com/documentation/100891/latest/ (visited on 13/05/2022).

[27] *RISC-V Vector Extension Intrinsics (v1.0)*. RISC-V Non-ISA Specifications, 16th November 2021. URL: https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/00882f19a84ab354dc8cf6a10c100b8daa2654e4/rvv-intrinsic-api.md (visited on 16/11/2021).

[28] Arm Ltd. *ARM C Language Extensions for SVE 0.0bet6*. 00bet6. 2020. URL: https://developer.arm.com/documentation/100987/0000/ (visited on 13/05/2022).

[29] *RISC-V ABIs Specification v1.0rc2*. 6th April 2022. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/v1.0-rc2/riscv-abi.pdf.

[30] Hongyan Xia et al. “CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety”. In: (2019), p. 14. DOI: 10/gm9ngg.

[31] Shiva Chen and Hsiangkai Wang. “Compiler Support For Linker Relaxation in RISC-V” (RISC-V Workshop Taiwan). 13th March 2019. URL: https://riscv.org/wp-content/uploads/2019/03/11.15-Shiva-Chen-Compiler-Support-For-Linker-Relaxation-in-RISC-V-2019-03-13.pdf (visited on 04/05/2022).

APPENDIX A

RUST_CHERI_COMPRESSED_CAP DOCUMENTATION

This section reproduces the documentation for `cheri-compressed-cap` C structs and functions that were linked into Rust. The full documentation[1] explains other Rust-specific decisions and implementation details.

[1] Available in **(redacted for anonymity)** or attached to submission

Struct rust_cheri_compressed_cap::CcxCap

```
#[repr(C, align(16))]
pub struct CcxCap<T: CompressedCapability> {
    _cr_cursor: T::Addr,
    cr_pesbt: T::Addr,
    _cr_top: T::FfiLength,
    cr_base: T::Addr,
    cr_tag: u8,
    cr_bounds_valid: u8,
    cr_exp: u8,
    cr_extra: u8,
}
```

Structure matching the C type `_cc_N(cap)`. Field order and layout is binary-compatible with the C version, assuming the C preprocessor macro `_CC_REVERSE_PESBT_CURSOR_ORDER` is *not* defined.

This is a plain-old-data type. It only supplies getters and setters, and does *not* guarantee any safety/correctness. For example, there are no added assertions or checks if you set the cursor to a value outside the bounds. However, the C FFI functions from CompressedCapability may have their own asserts. These are documented where possible.

For a safe interface, use one of the crate::wrappers

Fields

`_cr_cursor: T::Addr`

The bottom half of the capability as stored in memory.

If Self::cr_tag is 1, this is the capability's "cursor" i.e. the address it's actually pointing to.

`cr_pesbt: T::Addr`

The top half of the capability as stored in memory.

If Self::cr_tag is 1, this is the compressed capability metadata (permissions, otype, bounds, etc.).

`_cr_top: T::FfiLength`

The top of this capability's valid address range. Derived from Self::cr_pesbt. As long as Self::cr_tag is 1, the getter/setter will ensure it matches.

`cr_base: T::Addr`

The base of this capability's valid address range. Derived from Self::cr_pesbt. As long as Self::cr_tag is 1, the getter/setter will ensure it matches.

`cr_tag: u8`

Tag - if 1, this is a valid capability, 0 it's just plain data

`cr_bounds_valid: u8`

0 (false) if the bounds decode step was given an invalid capability. Should be 1 (true) for all non-Morello capabilities.

`cr_exp: u8`

The exponent used for storing the bounds. Stored from various places, only used in Morello-exclusive function cap_bounds_uses_value().

`cr_extra: u8`

"Additional data stored by the caller." Seemingly completely unused, essentially padding.

Implementations

```
impl<T: CompressedCapability> CcxCap<T>
```
[src]

Implements getters and setters similar to the C++-only member functions in the header.

```
pub fn reg_representation(&self) -> (bool, [T::Addr; 2])
```
[src]

Returns a `(tag, [cursor, pesbt])` tuple that represents all data required to store a capability in a register.

To store capabilities in memory, see Self::mem_representation

```
pub fn mem_representation(&self) -> (bool, [T::Addr; 2])
```
[src]

Returns a `(tag, [cursor, pesbt])` tuple that represents all data required to store a capability in memory.

To store capabilities in a register, see Self::reg_representation

```
pub fn tag(&self) -> bool
```
[src]

```
pub fn set_tag(&mut self, tag: bool)
```
[src]

```
pub fn base(&self) -> T::Addr
```
[src]

```
pub fn top(&self) -> T::Length                                    [src]

pub fn bounds(&self) -> (T::Addr, T::Length)                      [src]

pub fn set_bounds_unchecked(                                      [src]
    &mut self,
    req_base: T::Addr,
    req_top: T::Length
) -> bool
```

Sets the base and top of this capability using C FFI function CompressedCapability::set_bounds. Updates the PESBT field correspondingly. On non-Morello platforms, will fail with an assertion error if Self::tag() is not set.

```
pub fn address(&self) -> T::Addr                                  [src]

pub fn set_address_unchecked(&mut self, addr: T::Addr)            [src]

pub fn offset(&self) -> T::Offset                                 [src]

pub fn length(&self) -> T::Length                                 [src]

pub fn software_permissions(&self) -> u32                         [src]

pub fn set_software_permissions(&mut self, uperms: u32)           [src]

pub fn permissions(&self) -> u32                                  [src]

pub fn set_permissions(&mut self, perms: u32)                     [src]

pub fn otype(&self) -> u32                                        [src]

pub fn is_sealed(&self) -> bool                                   [src]

pub fn set_otype(&mut self, otype: u32)                           [src]

pub fn reserved_bits(&self) -> u8                                 [src]

pub fn set_reserved_bits(&mut self, bits: u8)                     [src]

pub fn flags(&self) -> u8                                         [src]

pub fn set_flags(&mut self, flags: u8)                            [src]

pub fn is_exact(&self) -> bool                                    [src]
```

Helper function for easily calling FFI function CompressedCapability::is_representable_cap_exact on this capability. Assertions are present in the C code, but should never be triggered.

```
pub fn is_representable_with_new_addr(&self, new_addr: T::Addr) [src]
```

> bool

Helper function for easily calling FFI function CompressedCapability::is_representable_new_addr on this capability. Assertions are present in the C code, but should never be triggered.

Trait Implementations

-] `impl<T: Clone + CompressedCapability> Clone for CcxCap<T>` [src]

```
where
    T::Addr: Clone,
    T::Addr: Clone,
    T::FfiLength: Clone,
    T::Addr: Clone,
```

-] `impl<T: CompressedCapability> Debug for CcxCap<T>` [src]

Debug printer for capabilities that decodes the PESBT field instead of printing it raw.

-] `impl<T: CompressedCapability> Default for CcxCap<T>` [src]

Equivalent to initialization pattern used in tests:

```
ccx_cap_t value;
memset(&value, 0, sizeof(value));
```

cc64.rs doesn't pick it up when it was automatically #derive-d, so it's manually implemented here

-] `impl<T: CompressedCapability> PartialEq<CcxCap<T>> for CcxCap<T>` [src]

Implements `operator==` from cheri_compressed_cap_common.h

`impl<T: Copy + CompressedCapability> Copy for CcxCap<T>` [src]

```
where
    T::Addr: Copy,
    T::Addr: Copy,
    T::FfiLength: Copy,
    T::Addr: Copy,
```

`impl<T: CompressedCapability> Eq for CcxCap<T>` [src]

Auto Trait Implementations

```
impl<T> RefUnwindSafe for CcxCap<T>
where
    <T as CompressedCapability>::Addr: RefUnwindSafe,
    <T as CompressedCapability>::FfiLength: RefUnwindSafe,

impl<T> Send for CcxCap<T>
```

```
ere
    <T as CompressedCapability>::Addr: Send,
    <T as CompressedCapability>::FfiLength: Send,

impl<T> Sync for CcxCap<T>
where
    <T as CompressedCapability>::Addr: Sync,
    <T as CompressedCapability>::FfiLength: Sync,

impl<T> Unpin for CcxCap<T>
where
    <T as CompressedCapability>::Addr: Unpin,
    <T as CompressedCapability>::FfiLength: Unpin,

impl<T> UnwindSafe for CcxCap<T>
where
    <T as CompressedCapability>::Addr: UnwindSafe,
    <T as CompressedCapability>::FfiLength: UnwindSafe,
```

Blanket Implementations

-] `impl<T> Any for T` [src]

```
where
    T: 'static + ?Sized,
```

-] `impl<T> Borrow<T> for T` [src]

```
where
    T: ?Sized,
```

-] `impl<T> BorrowMut<T> for T` [src]

```
where
    T: ?Sized,
```

-] `impl<T> From<T> for T` [src]

-] `impl<T, U> Into<U> for T` [src]

```
where
    U: From<T>,
```

-] `impl<T> ToOwned for T` [src]

```
where
    T: Clone,
```

```
type Owned = T
```

The resulting type after obtaining ownership.

-] `impl<T, U> TryFrom<U> for T` [src]

```
where
    U: Into<T>,
```

```
type Error = Infallible
```

The type returned in the event of a conversion error.

```
mpl<T, U> TryInto<U> for T                                    [src]
ere
  U: TryFrom<T>,
```

```
type Error = <U as TryFrom<T>>::Error
```

The type returned in the event of a conversion error.

Trait CompressedCapability

Trait rust_cheri_compressed_cap::CompressedCapability

```
pub trait CompressedCapability: Sized + Copy + Clone {
    type Length: NumType + From<Self::Addr>;
    type Offset: NumType + From<Self::Addr>;
    type Addr: NumType + Into<Self::Offset> + Into<Self::Length>;
    type FfiLength: FfiNumType<Self::Length>;
    type FfiOffset: FfiNumType<Self::Offset>;
[+] Show associated constants and methods
}
```

Trait defining an Rust version of the public API for a specific capability type. A type X implementing CompressedCapability is equivalent to the API provided by `cheri_compressed_cap_X.h` in C, where `ccx_cap_t` is equivalent to CcxCap.

It is not recommended to call the trait functions directly. Instead, use one of the crate::wrappers.

Associated Types

`type Length: NumType + From<Self::Addr>` [src]

ccx_length_t Rust-land equivalent - should be a superset of Addr

`type Offset: NumType + From<Self::Addr>` [src]

ccx_offset_t Rust-land equivalent - should be a superset of Addr

`type Addr: NumType + Into<Self::Offset> + Into<Self::Length>` [src]

ccx_addr_t equivalent

`type FfiLength: FfiNumType<Self::Length>` [src]

ccx_length_t C-land equivalent - should have a memory layout identical to the C ccx_length_t. This is separate from Length because for 128-bit types the Rust and C versions may not look the same. In practice, we just assume they are the same (see crate::c_funcs documentation).

`type FfiOffset: FfiNumType<Self::Offset>` [src]

ccx_offset_t C-land equivalent - should have a memory layout identical to the C

ccx_offset_t. See Self::FfiLength for an explanation.

Associated Constants

`const PERM_GLOBAL: u32` [src]

CCX_PERM_GLOBAL equivalent These are the same for 64 and 128bit, but should be overridden for Morello-128

`const PERM_EXECUTE: u32` [src]

`const PERM_LOAD: u32` [src]

`const PERM_STORE: u32` [src]

`const PERM_LOAD_CAP: u32` [src]

`const PERM_STORE_CAP: u32` [src]

`const PERM_STORE_LOCAL: u32` [src]

`const PERM_SEAL: u32` [src]

`const PERM_CINVOKE: u32` [src]

`const PERM_UNSEAL: u32` [src]

`const PERM_ACCESS_SYS_REGS: u32` [src]

`const PERM_SETCID: u32` [src]

`const MAX_REPRESENTABLE_OTYPE: u32` [src]

`const OTYPE_UNSEALED: u32` [src]

CCX_OTYPE_UNSEALED equivalent

`const OTYPE_SENTRY: u32` [src]

`const OTYPE_RESERVED2: u32` [src]

`const OTYPE_RESERVED3: u32` [src]

`const MAX_UNRESERVED_OTYPE: u32` [src]

Required methods

`fn compress_raw(src_cap: &CcxCap<Self>) -> Self::Addr` [src]

Generate the `pesbt` bits for a capability (the top bits, which encode permissions, object type, compressed bounds, etc.) This transformation can be undone with

Self::decompress_raw.

This is presumably intended for storing compressed capabilities in e.g. registers. Its counterpart for storing compressed capabilities in memory is Self::compress_mem.

```
fn decompress_raw(
    pesbt: Self::Addr,
    cursor: Self::Addr,
    tag: bool
) -> CcxCap<Self>
```
[src]

Decompress a (pesbt, cursor) pair into a capability. This transformation can be undone with Self::compress_raw.

```
fn compress_mem(src_cap: &CcxCap<Self>) -> Self::Addr
```
[src]

Generate the `pesbt` bits for a capability (the top bits, which encode permissions, object type, compressed bounds, etc.) This transformation can be undone with Self::decompress_mem.

This is presumably intended for storing compressed capabilities in memory. It is equivalent to calling Self::compress_raw and XOR-ing the result with a "null mask". Presumably this transformation prevents all-zero data from being interpreted as a capability?

```
fn decompress_mem(
    pesbt: Self::Addr,
    cursor: Self::Addr,
    tag: bool
) -> CcxCap<Self>
```
[src]

Decompress a (pesbt, cursor) pair into a capability. This transformation can be undone with Self::compress_mem.

This is equivalent to XOR-ing the pesbt with a "null mask" and calling Self::decompress_raw. Presumably the null mask prevents all-zero data from being interpreted as a capability?

```
fn get_uperms(cap: &CcxCap<Self>) -> u32
```
[src]

Gets the user/software-defined permissions from the CcxCap::cr_pesbt field

Counterpart: Self::update_uperms

```
fn get_perms(cap: &CcxCap<Self>) -> u32
```
[src]

Gets the hardware-defined permissions from the CcxCap::cr_pesbt field

Counterpart: Self::update_perms

```
fn get_otype(cap: &CcxCap<Self>) -> u32
```
[src]

Gets the object type from the CcxCap::cr_pesbt field

Counterpart: Self::update_otype

```
fn get_reserved(cap: &CcxCap<Self>) -> u8
```
[src]

Gets the reserved bits from the CcxCap::cr_pesbt field

Counterpart: Self::update_reserved

```
fn get_flags(cap: &CcxCap<Self>) -> u8
```
[src]

Gets the flags from the CcxCap::cr_pesbt field

Counterpart: Self::update_flags

```
fn update_uperms(cap: &mut CcxCap<Self>, value: u32)
```
[src]

Updates the user/software-defined permissions field in CcxCap::cr_pesbt

Counterpart: Self::get_uperms

```
fn update_perms(cap: &mut CcxCap<Self>, value: u32)
```
[src]

Updates the hardware-defined permissions field in CcxCap::cr_pesbt

Counterpart: Self::get_perms

```
fn update_otype(cap: &mut CcxCap<Self>, value: u32)
```
[src]

Updates the object type field in CcxCap::cr_pesbt

Counterpart: Self::get_otype

```
fn update_reserved(cap: &mut CcxCap<Self>, value: u8)
```
[src]

Updates the reserved field in CcxCap::cr_pesbt

Counterpart: Self::get_reserved

```
fn update_flags(cap: &mut CcxCap<Self>, value: u8)
```
[src]

Updates the flags field in CcxCap::cr_pesbt

Counterpart: Self::get_flags

```
fn extract_bounds_bits(pesbt: Self::Addr) -> CcxBoundsBits
```
[src]

Extracts the floating-point encoded bounds from CcxCap::cr_pesbt

```
fn set_bounds(
    cap: &mut CcxCap<Self>,
    req_base: Self::Addr,
    req_top: Self::Length
) -> bool
```
[src]

Sets the capability bounds to bounds that encompass (req_base, req_top). Because a floating-point representation is used for bounds, it may not be able to set (req_base,

req_top) exactly. In this case it will return False.

Updates CcxCap::cr_pesbt, CcxCap::_cr_top, CcxCap::cr_base

```
fn is_representable_cap_exact(cap: &CcxCap<Self>) -> bool
```
[src]

Check if the range (CcxCap::cr_base, CcxCap::_cr_top) can be encoded exactly with the floating-point encoding

```
fn is_representable_new_addr(
    sealed: bool,
    base: Self::Addr,
    length: Self::Length,
    cursor: Self::Addr,
    new_cursor: Self::Addr
) -> bool
```
[src]

Check if a capability with the parameters `sealed, base, length, cursor` would be representable if the cursor were updated to `new_cursor`.

```
fn make_max_perms_cap(
    base: Self::Addr,
    cursor: Self::Addr,
    top: Self::Length
) -> CcxCap<Self>
```
[src]

Generate a capability for `base, top, cursor` with the maximum available permissions

```
fn get_representable_length(length: Self::Length) ->
Self::Length
```
[src]

Get the minimum representable length greater than or equal to `length`.

If `get_representable_length(l) == l` then bounds of length `l` are exactly representable (if properly aligned).

See also Self::get_required_alignment, Self::get_alignment_mask.

```
fn get_required_alignment(length: Self::Length) -> Self::Length
```
[src]

Get the alignment required for bounds of some `length` to be exactly represented.

See also Self::get_representable_length, Self::get_alignment_mask.

```
fn get_alignment_mask(length: Self::Length) -> Self::Length
```
[src]

Get a mask which aligns a bounds of some `length` to be exactly representable.

See also Self::get_representable_length, Self::get_required_alignment.

Implementors

```
impl CompressedCapability for Cc64                                    [src]

    type Length = u64

    type Offset = i64

    type Addr = u32

    type FfiLength = u64

    type FfiOffset = i64

    const MAX_REPRESENTABLE_OTYPE: u32                                [src]

    _CC_N(OTYPE_UNSEALED_SIGNED) = (((int64_t)-1) - 0u)```
    The OTYPE field is 4 bits (50:47) in CC64

[−] impl CompressedCapability for Cc128                               [src]

    type Length = u128

    type Offset = i128

    type Addr = u64

    type FfiLength = u128

    type FfiOffset = i128

    const MAX_REPRESENTABLE_OTYPE: u32                                [src]
```

The OTYPE field is 18 bits (108:91) in CC128

APPENDIX B

Code Snippets

This appendix contains code snippets referenced in the document. Some small snippets include comparison to generated assembly, which is extracted using godbolt.org. The full codebases are available online at **(redacted for anonymity)**, and attached to the submission.

B.1 C example — Basic RVV program

This a reproduction of https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/master/examples/rvv_memcpy.c, with annotations to mirror the steps from Section 4.1.3.

```c
#include <riscv_vector.h>

void *memcpy_vec(void *dst, void *src, size_t n) {
  void *save = dst;
  // copy data byte by byte
  for (size_t vl; n > 0; n -= vl, src += vl, dst += vl) {
    // Use a vsetvl intrinsic to get the
    // vector length for this iteration.
    vl = vsetvl_e8m8(n);

    // Allocate vector registers by declaring
    // variables with vector types
    vuint8m8_t vec_src;

    // Pass the vector length to intrinsics
    vec_src = vle8_v_u8m8(src, vl);
    vse8_v_u8m8(dst, vec_src, vl);
  }
  return save;
}
```

B.2 C example — Saving/restoring vector registers

This is an example of a compiler generating code to save/restore vector registers before/after calling another function. Generated from Clang 15 — output availble from https://godbolt.org/z/4xfMoxsT7

Figure B.1: Saving/restoring vector registers

(a) C code

```c
void some_other_function();

void vector_memcpy(
    size_t n,
    const uint16_t* in,
    uint16_t* out
) {
    size_t vl =
    vsetvl_e16m8(n);

    vuint16m8_t data =
    vle16_v_u16m8(in, vl);
    some_other_function();
    vse16_v_u16m8(
    out, data, vl
    );
}
```

(b) Generated assembly

```
vector_memcpy:
# Preamble
    addi    sp, sp, -64
    sd      ra, 56(sp)
    sd      s0, 48(sp)
    sd      s1, 40(sp)
# Allocate stack for VLEN*8
    csrr    a3, vlenb
    slli    a3, a3, 3
    sub     sp, sp, a3
# size_t vl = vsetvl_e16m8(n);
    vsetvli s1, a0, e16, m8, ta, mu
# vuint16m8_t data = ...;
    vle16.v v8, (a1)
# Save data to stack
    addi    a0, sp, 32
    vs8r.v  v8, (a0)
    mv      s0, a2
# some_other_function();
    call    some_other_function@plt
# Reload data from stack
    vsetvli zero, s1, e16, m8, ta,
    ↪ mu
    addi    a0, sp, 32
    vl8re8.v    v8, (a0)
# vse16_v_u16m8(...);
    vse16.v v8, (s0)
# Postamble, deallocate VLEN*8
    csrr    a0, vlenb
    slli    a0, a0, 3
    add     sp, sp, a0
    ld      ra, 56(sp)
    ld      s0, 48(sp)
    ld      s1, 40(sp)
    addi    sp, sp, 64
    ret
```

B.3 C example — Arm SVE

This section shows how to use Arm SVE in C, in various ways. The output is compiled with the latest GCC (Clang also supports Arm SVE, but generates longer output). Output for GCC and Clang is available at https://godbolt.org/z/8edWMscfP.

Figure B.2: Arm SVE — Autovectorization

```c
void test_auto_vector(int64_t* data, int64_t n) {
    for (int i = 0; i < n; i++) {
        data[i] += 1;
    }
}
```

(a) C code

```
test_auto_vector:
        cmp     x1, 0
        ble     .L1
        mov     w3, w1
        mov     x2, 0
        cntd    x4
        whilelo p0.d, wzr, w1
.L3:
        ld1d    z0.d, p0/z, [x0, x2, lsl 3]
        add     z0.d, z0.d, #1
        st1d    z0.d, p0, [x0, x2, lsl 3]
        add     x2, x2, x4
        whilelo p0.d, w2, w3
        b.any   .L3
.L1:
        ret
```

(b) Generated assembly

Figure B.3: Arm SVE — Intrinsics

```c
void test_intrinsics_sve(int64_t* data, int64_t n) {
    if (n == 0) return;
    int64_t i = 0;
    svbool_t pg = svwhilelt_b64(i, n);
    svint64_t one = svdup_s64(1);
    do {
        svint64_t d_vec = svld1(pg, &data[i]);
        svst1(pg, &data[i], svadd_z(pg, d_vec, one));
        i += svcntd();
        pg = svwhilelt_b64(i, n);
    }
    while (svptest_any(svptrue_b64(), pg));
}
```

(a) C code

```asm
test_intrinsics_sve:
        cbz     x1, .L6
        mov     x2, 0
        cntd    x3
        whilelt p0.d, xzr, x1
        mov     z1.d, #1
.L8:
        ld1d    z0.d, p0/z, [x0, x2, lsl 3]
        movprfx z0.d, p0/z, z0.d
        add     z0.d, p0/m, z0.d, z1.d
        st1d    z0.d, p0, [x0, x2, lsl 3]
        add     x2, x2, x3
        whilelt p0.d, x2, x1
        b.any   .L8
.L6:
        ret
```

(b) Generated assembly

Figure B.4: Arm SVE — Inline Assembly

```c
void test_asm_sve(int64_t* data, int64_t n) {
    if (n == 0) return;
    int64_t i = 0;

    asm ("whilelt p0.d, %0, %1" :: "r"(i), "r"(n));
    asm ("mov z0.d, #1");
    asm("loop:");
    {
        // svint64_t d_vec = svld1(pg, &data[i]);
        // Load data[i] -> z1.d, where i = 64-bit index (3 bit shift
        ↪  up from 8-bit)
        asm ("ld1d z1.d, p0/z, [%0, %1, lsl #3]" :: "r"(data),
        ↪  "r"(i));
        // Add z1 + z0, storing the results in z1, where p0/Masks the
        ↪  addition
        asm ("add z1.d, p0/m, z1.d, z0.d");
        asm ("st1d z1.d, p0, [%0, %1, lsl #3]" :: "r"(data), "r"(i) :
        ↪  "memory");
        // svst1(pg, &data[i], svadd_z(pg, d_vec, one));
        i += svcntd();
        // This sets the Z flag to 1 if nothing is left
        asm ("whilelt p0.d, %0, %1" :: "r"(i), "r"(n));
        // b.ne = Not Equal = (if Z flag is 0)
        // if Z flag is 0 i.e. something is left, jump to loop
        asm ("b.ne loop");
    }
}
```

(a) C code

```asm
test_asm_sve:
        cbz     x1, .L13
        mov     x2, 0
        whilelt p0.d, x2, x1
        mov z0.d, #1
loop:
        ld1d z1.d, p0/z, [x0, x2, lsl #3]
        add z1.d, p0/m, z1.d, z0.d
        st1d z1.d, p0, [x0, x2, lsl #3]
        cntd    x0
        whilelt p0.d, x0, x1
        b.ne loop
.L13:
        ret
```

(b) Generated assembly

B.4 `riscv-v-lite` — Vector memory accesses

```rust
/// Converts a decoded memory operation to the list of accesses it performs.
fn get_load_store_accesses(&mut self, rd: u8,
    addr_p: (u64, Provenance),
    rs2: u8, vm: bool, op: DecodedMemOp)
    -> Result<Vec<(VectorElem, u64)>> {
  // Vector of (VectorElem, Address)
  let mut map = vec![];

  let (base_addr, _) = addr_p;

  use DecodedMemOp::*;
  match op {
    Strided{stride, evl, nf, eew, emul, ..} => {
      // For each segment
      for i_segment in self.vstart..evl {
        let seg_addr = base_addr + (i_segment as u64 * stride);

        // If we aren't masked out...
        if !self.vreg.seg_masked_out(vm, i_segment) {
          // For each field
          let mut field_addr = seg_addr;
          for i_field in 0..nf {
            // ... perform the access
            let vec_elem = VectorElem::check_with_lmul(
              // Register group start
              // For field 0, = rd
              // For field 1, = rd + (number of registers/group)
              // etc.
              rd + (i_field * emul.num_registers_consumed()),
              eew, emul,
              // Element index within register group
              i_segment
            );
            map.push((vec_elem, field_addr));
            // and increment the address
```

```
            field_addr += eew.width_in_bytes();
          }
        }
      }
    }
    FaultOnlyFirst{evl, nf, eew, emul} => {
      // We don't handle the exceptions here
      // This just lists the accesses that will be attempted
      // This is exactly the same code as for Strided, but it
↪   calculates the stride
      let stride = eew.width_in_bytes() * (nf as u64);

      // For each segment
      for i_segment in self.vstart..evl {
        let seg_addr = base_addr + (i_segment as u64 * stride);

        // If we aren't masked out...
        if !self.vreg.seg_masked_out(vm, i_segment) {
          // For each field
          let mut field_addr = seg_addr;
          for i_field in 0..nf {
            // ... perform the access
            let vec_elem = VectorElem::check_with_lmul(
              rd + (i_field * emul.num_registers_consumed()),
              eew, emul,
              i_segment
            );
            map.push((vec_elem, field_addr));
            // and increment the address
            field_addr += eew.width_in_bytes();
          }
        }
      }
    }
    Indexed{index_ew, evl, nf, eew, emul, ..} => {
      // i = element index in logical vector (which includes groups)
      for i_segment in self.vstart..evl {
```

```
        // Get our index
        let seg_offset = self.vreg.load_vreg_elem_int(index_ew, rs2,
↪   i_segment)?;
        let seg_addr = base_addr + seg_offset as u64;

        // If we aren't masked out...
        if !self.vreg.seg_masked_out(vm, i_segment) {
          // For each field
          let mut field_addr = seg_addr;
          for i_field in 0..nf {
            // ... perform the access
            let vec_elem = VectorElem::check_with_lmul(
              rd + (i_field * emul.num_registers_consumed()),
              eew, emul,
              i_segment
            );
            map.push((vec_elem, field_addr));
            // and increment the address
            field_addr += eew.width_in_bytes();
          }
        }
      }
    }
    WholeRegister{num_regs, eew, ..} => {
      if vm == false {
        // There are no masked variants of this instruction
        bail!("WholeRegister operations cannot be masked")
      }

      let mut addr = base_addr;
      let vl = op.evl();
      // For element in register set...
      for i in self.vstart..vl {
        // ...perform the access
        let vec_elem = VectorElem::check_with_num_regs(
          rd,
          eew, num_regs,
```

```
            i as u32
          );
          map.push((vec_elem, addr));
          addr += eew.width_in_bytes();
        }
      }
      ByteMask{evl, ..} => {
        if vm == false {
          // vlm, vsm cannot be masked out
          bail!("ByteMask operations cannot be masked")
        }

        let mut addr = base_addr;
        // evl = number of 8-bit elements required for the mask
        // self.vstart = in terms of bytes
        for i in self.vstart..evl {
          let vec_elem = VectorElem::check_with_lmul(
            rd,
            Sew::e8, Lmul::e1,
            i
          );
          map.push((vec_elem, addr));
          addr += 1;
        }
      }
    };

    Ok(map)
}
```

APPENDIX C

Fast path vector checks

This appendix describes methods of calculating tight bounds for vector memory accesses (Section 3.1.2.2) and ways that architectural complexity can be traded off to calculate *wider* bounds. These methods calculate the entire bounds up front, and while they are used in the emulator a hardware implementation may find it introduces too much latency.

C.1 Masked accesses

For all masked accesses, masked-out/inactive segments should not trigger capability exceptions. Therefore, a tight bounds must include only the smallest and largest active segments. These segments can be found by inspecting the mask vector: either checking each bit in turn or using parallel logic to find the lowest/highest set bits. Care must be taken with these checks to ensure elements outside the range $[$*vstart*, *evl*$)$ are not counted.

$$\texttt{vstart}_{active} = \min(i \; \forall \; \texttt{vstart} \leq i < \texttt{evl} \text{ where } mask[i] = 1) \tag{C.1.1}$$

$$\texttt{evl}_{active} = \max(i \; \forall \; \texttt{vstart} \leq i < \texttt{evl} \text{ where } mask[i] = 1) + 1 \tag{C.1.2}$$

Tradeoffs

If using parallel logic to find the lowest/highest bits, it could be difficult to account for $[$*vstart*, *evl*$)$. An implementation could choose to only calculate tight bounds when the mask is fully utilized, i.e. $\textit{vstart} = 0$, $\textit{evl} = \textit{VLEN}$, and assume wider bounds otherwise.

Accounting for masked accesses at all may not be worth the extra complexity. Only elements masked off on the edges make any difference, and it may be uncommon for long runs of edge elements to be masked off. Thus, an implementation could choose to ignore masking entirely when computing the ranges. This does mean that all failures become Likely-Failure when masking is enabled, because all elements outside the capability bounds may be masked off.

C.2 Unit accesses

For unit segmented accesses, which includes fault-only first, the tight address range for an access is simple to calculate. Whole register and bytemask accesses can simplify this by fixing `nf = 1` and `eew = 8`.

$$\texttt{base} + [\texttt{vstart}_{active} * \texttt{nf} * \texttt{eew}, \texttt{evl}_{active} * \texttt{nf} * \texttt{eew}) \tag{C.2.1}$$

Tradeoffs

`nf` is not guaranteed to be a power of two (except for the whole-register case), so calculating the 'tight' address range would require a multiplication by an arbitrary four-bit value between 1 and 8. If this multiplication is too expensive, implementations could choose to classify all $\texttt{nf} > 1$ cases as Unchecked.

Unless extra restrictions are placed on `vstart`, calculating the start of this range requires another arbitrary multiplication. To avoid this one could assume `vstart = 0` and treat failures as Likely-Failure for other cases. Once could also classify all nonzero `vstart` accesses as Unchecked.

Even if the previous two optimizations are applied, the final range still requires a multiplication $\texttt{evl} * \texttt{eew}$. Thankfully, because `eew` may only be one of four powers-of-two, this can be encoded as a simple shift.

C.3 Strided accesses

Strided accesses bring further complication, especially as the stride may be negative.

$$\texttt{base} + \begin{cases} [\texttt{vstart}_{active} * \texttt{stride}, (\texttt{evl}_{active} - 1) * \texttt{stride} + \texttt{nf} * \texttt{eew}) & \texttt{stride} \geq 0 \\ [(\texttt{evl}_{active} - 1) * \texttt{stride}, \texttt{vstart}_{active} * \texttt{stride} + \texttt{nf} * \texttt{eew}) & \texttt{stride} < 0 \end{cases} \tag{C.3.1}$$

This is formed of three components:

- $\texttt{vstart}_{active} * \texttt{stride}$, the start of the first segment. This can be simplified to 0, just like for unit accesses, to avoid an arbitrary multiplication.
- $(\texttt{evl}_{active} - 1) * \texttt{stride}$, the start of the final segment. This requires an arbitrary multiplication, unless strided accesses are all Unchecked.
- $\texttt{nf} * \texttt{eew}$, the length of a segment, which can be implemented with a shift.

C.4 Indexed accesses

This is the most complicated access of the bunch, because the addresses cannot be computed without reading the index register.

$$[\texttt{base} + \min(\texttt{offsets}[\texttt{vstart}_{active}..\texttt{evl}_{active}]), \tag{C.4.1}$$

$$\texttt{base} + \max(\texttt{offsets}[\texttt{vstart}_{active}..\texttt{evl}_{active}]) + \texttt{nf} * \texttt{eew}) \tag{C.4.2}$$

The most expensive components here are of course min, max of the offsets. These could be calculated in hardware through parallel reductions, making it slightly more efficient than looping over each element. A low-hanging optimization could be to remove the $\texttt{vstart}_{active}..\texttt{evl}_{active}$ range condition, performing the reduction over the whole register group, which would make failures Likely-Failure where $\texttt{vstart}_{active} != 0 \,||\, \texttt{evl}_{active} != \texttt{VLMAX}$. This calculation could also be restricted to certain register configurations to reduce the amount of required hardware. Indeed, the amount of hardware could be reduced to zero by simply classifying all indexed accesses as Unchecked.

APPENDIX D

Compiler information

D.1 Vanilla RVV command-line options

Compiler	Required Arguments	Notes
Clang-13	`-march=rv64gv0p10` `-menable-experimental-extensions`	Supports intrinsics, inline assembly for RVV v0.1
Clang-14+	`-march=rv64gv`	Supports intrinsics, inline assembly for RVV v1.0
GCC 10.1	`-march=rv64g_v`	Requires special toolchain (see Appendix D.5) and has incomplete support (see Appendix D.3)

Table D.1: Command-line arguments for compiling RVV code on non-CHERI compilers (assuming the base ISA is `rv64g`)

D.2 CHERI-RVV command-line options

Compiler	Required Arguments	Notes

CHERI Clang-13	`-march=rv64gv0p10xcheri`
	`-menable-experimental-extensions`
	`-mabi=l64pc128`
	`-mno-relax`
		Supports intrinsics, inline assembly for RVV v0.1
		ABI string sets capability width.
		Must disable linker relaxations.

Table D.2: Command-line arguments for compiling CHERI-RVV code (assuming the base ISA is `rv64g`)

By default CHERI-Clang doesn't actually compile capability-enabled code. The documentation on enabling capabilities is unfortunately sparse and outdated. In particular, the

CHERI-Clang help menu states that `--cheri` will "Enable CHERI support with the default capability size", but this has no effect (at least on RISC-V). To find up-to-date answers, we consulted the source code for the CHERIbuild build tool[1].

CHERIbuild's code[2] revealed three requirements:

- The architecture string must contain `xcheri`
- The capability length must be set using the ABI string
  - In pure-capability mode, pointers and capabilities are `CLEN` long
    - Example string: `l64pc128`
    - Integer width (`long`, or `l`) = XLEN = 64-bits
    - Pointer width (`p`) = Capability width (`p`) = CLEN = 128-bits
  - For hybrid mode, pointers remain `XLEN` long and capability length is not specified
    - Example string: `lp32`
    - Integer width (`l`) = XLEN = Pointer width (`p`) = 32-bits
- "Linker relaxations", where function calls are converted to short jumps[31], must be disabled.
  - This is likely because CHERI requires function calls to go through capabilities
  - However the code that adds this option wasn't documented, so there may be more to it

Once the above options are set, plain CHERI-RISC-V code compiles without a hitch. Changes to CHERI-Clang itself are required to compile vectors (Section 4.2.1).

D.3 Compiler support for RVV

While most compilers support all memory access archetypes, there are a few notable exceptions. GCC has the most: there is no support for fractional LMUL or bytemask accesses, the intrinsics for segmented accesses are named differently, and fault-only-first intrinsics emit incorrect instructions[3]. GCC RVV support has been deprioritized in favor of LLVM[4], so the rough edges make sense. LLVM-13-based compilers (including CHERI-Clang) support all specified archetypes except bytemask accesses. CHERI-Clang doesn't support intrinsics, but all inline assembly support is intact. Support for bytemask accesses is only available in LLVM-14 and up.

[1] `CSTRD-CHERI/cheribuild` on Github
[2] `config/compilation_targets.py:176` in `CSTRD-CHERI/cheribuild` on GitHub
[3] On GCC, fault-only-first intrinsics seem to emit `vsetvli`.
[4] `https://github.com/riscv-collab/riscv-gcc/issues/320`

D.4 Ensuring compatibility between different compilers

The `vector_memcpy` test program uses the preprocessor to identify the current compiler and how that compiler supports various vector instructions. It is reproduced here in case it can be useful for other vector-agnostic programs.

```
#define ASM_PREG(val) "r"(val)
// GCC doesn't like __has_feature(capabilities), so define a
↪  convenience value
// which is only 1 when in LLVM with __has_feature(capabilities)
#define HAS_CAPABILITIES 0

// Patch over differences between GCC, clang, and CHERI-clang
#if defined(__llvm__)
// Clang intrinsics are correct for segmented loads,
// and supports fractional LMUL.
// Clang 14+ has the correct intrinsics for bytemask loads,
// and Clang has been tested with wholereg ASM

    // Use intrinsics for BYTEMASK in newer Clangs,
    // otherwise the intrinsics don't exist
    #if __clang_major__ >= 14
        #define ENABLE_BYTEMASK 1
        #define USE_ASM_FOR_BYTEMASK 0
    #else
        // LLVM 13 does not support bytemask
        #define ENABLE_BYTEMASK 0
    #endif

    #if __has_feature(capabilities)
        #undef HAS_CAPABILITIES
        #define HAS_CAPABILITIES 1

        #if __has_feature(pure_capabilities)
            #undef ASM_PREG
            #define ASM_PREG(val) "C"(val)
        #endif
```

```
        // Enable everything
        #define ENABLE_UNIT 1
        #define ENABLE_STRIDED 1
        #define ENABLE_INDEXED 1
        #define ENABLE_MASKED 1
        #define ENABLE_SEGMENTED 1
        #define ENABLE_FRAC_LMUL 1
        #define ENABLE_ASM_WHOLEREG 1
        #define ENABLE_FAULTONLYFIRST 1
        // BYTEMASK is disabled above

        // Use ASM for everything
        #define USE_ASM_FOR_UNIT 1
        #define USE_ASM_FOR_STRIDED 1
        #define USE_ASM_FOR_INDEXED 1
        #define USE_ASM_FOR_MASKED 1
        #define USE_ASM_FOR_SEGMENTED 1
        // Wholereg has no intrinsics, always ASM
        #define USE_ASM_FOR_FAULTONLYFIRST 1
    #else
        // Enable everything
        #define ENABLE_UNIT 1
        #define ENABLE_STRIDED 1
        #define ENABLE_INDEXED 1
        #define ENABLE_MASKED 1
        #define ENABLE_SEGMENTED 1
        #define ENABLE_FRAC_LMUL 1
        #define ENABLE_ASM_WHOLEREG 1
        #define ENABLE_FAULTONLYFIRST 1

        // Use intrinsics for everything
        #define USE_ASM_FOR_UNIT 0
        #define USE_ASM_FOR_STRIDED 0
        #define USE_ASM_FOR_INDEXED 0
        #define USE_ASM_FOR_MASKED 0
        #define USE_ASM_FOR_SEGMENTED 0
        // Wholereg has no intrinsics, always ASM
```

```
        #define USE_ASM_FOR_FAULTONLYFIRST 0
    #endif
#elif defined(__GNUC__) && !defined(__INTEL_COMPILER)
// GNU exts enabled, not in LLVM or Intel, => in GCC

// GCC from RISC-V toolchain rvv-intrinsics branch
// has incorrect names for segmented intrinsics,
// doesn't support fractional LMUL,
// doesn't support byte-mask,
// emits incorrect code for fault-only-first intrinsics
//   (it seems to emit a vsetvli instruction).

    // Enable everything except fractional LMUL and bytemask
    #define ENABLE_UNIT 1
    #define ENABLE_STRIDED 1
    #define ENABLE_INDEXED 1
    #define ENABLE_MASKED 1
    #define ENABLE_SEGMENTED 1
    #define ENABLE_FRAC_LMUL 0
    #define ENABLE_BYTEMASK 0
    #define ENABLE_ASM_WHOLEREG 1
    #define ENABLE_FAULTONLYFIRST 1

    // Use intrinsics for all except segmented loads
    #define USE_ASM_FOR_UNIT 0
    #define USE_ASM_FOR_STRIDED 0
    #define USE_ASM_FOR_INDEXED 0
    #define USE_ASM_FOR_MASKED 0
    #define USE_ASM_FOR_SEGMENTED 1
    // bytemask is disabled
    #define USE_ASM_FOR_BYTEMASK 0
    // Wholereg is always ASM
    // fault-only-first intrinsics emit the wrong instruction
    #define USE_ASM_FOR_FAULTONLYFIRST 1
#endif
```

D.5 Building `riscv-gnu-toolchain` with vector support

As of May 2022, the RISC-V GNU toolchain (hosted at `riscv-collab/riscv-gnu-toolchain` on Github) does not support the vector extension or it's intrinsics. The `rvv-intrinsic` branch of this repository claimed to support vector intrinsics, but it was slightly outdated and has been deleted as of 17th May 2022. It referenced a repository for `glibc` that no longer exists as a submodule, which makes compilation impossible. We have archived this branch online (**(redacted for anonymity)**) and fixed that issue. This appendix describes how to build the toolchain.

To build the full toolchain with intrinsic support, perform the following steps (derived by the author independently, then amended based on macOS instructions from [5]):

1. Clone the repository itself:

```
$ git clone (redacted for anonymity)
```

2. Clone the `riscv-gcc` submodule:

```
$ git submodule update --init --recursive --progress --force ./riscv-gcc
```

   - On macOS, it may be necessary to disable SSL:

   ```
   $ git -c http.sslVerify=false submodule ...
   ```

3. Configure the compiler so it supports all General extensions, Compressed instructions, and Vector extension:

```
$ ./configure --prefix=<output directory> --with-arch=rv64gcv --with-abi=lp64d
```

4. Build the `newlib` version to compile for bare-metal platforms:

```
$ make newlib -j$(nproc)
```

[5] https://github.com/riscv-collab/riscv-gcc/issues/323

APPENDIX E

CHERI-RVV CHANGES FROM CHERI AND RVV

This appendix summarizes the differences between a CHERI-RVV ISA, as emulated by `riscv-v-lite`, and a CHERI-RISC-V ISA with the vanilla RVV extension.

E.1 Loading/storing with capabilities

All CHERI-RISC-V instructions are unchanged. The RVV memory access instructions (listed in Section 2.5) have their behaviour changed, and all other RVV memory access instructions are unchanged.

In capability mode, vector memory accesses are changed to use capabilities. The `rs1` field in the encoding for all memory access instructions is changed to `cs1`, which specifies the capability register holding the *base capability*. In integer mode, the value held in register `rs1` is added to the DDC to create the *base capability*. The base capability's *cursor* is used as the base address for all accesses.

The behaviour of each vector memory access is unchanged, except for exception behaviour. A synchronous exception is raised on element `i` of width `eew` at address `addr` if

- The base capability tag is not set.
- The base capability is sealed.
- The base capability does not have adequate permissions.
  - e.g. PERMIT_LOAD for loads,

    PERMIT_STORE for stores.
- `addr < base_capability.base`.
- `addr + eew > base_capability.top`.

All other exceptions that would be raised by a vanilla RVV equivalent, e.g. unaligned access exceptions, are also raised. Fault-only-first loads silently swallow capability-related exceptions when `i > 0`, setting `vl` instead of taking a trap, just like all other synchronous exceptions.

In a future version, it may be desirable for some capability exceptions to trap before any accesses are attempted.**[TODO1 Make sure this is made clear in main content]** For example, passing a sealed, untagged, or permissionless capability to a memory access usually reflects a serious programming error, and should not be ignored in any case. The current emulator would ignore those errors if all elements were masked out, and would silently swallow them in fault-only-first. Element-specific exceptions, i.e. bounds violations, should still always be swallowed by fault-only-first.

E.2 Capabilities-in-vectors changes

`vsetvl` instructions are modified to accept a `SEW` value mapping to 128-bit elements (Table E.1). The arithmetic RVV instructions implemented by the emulator, listed in Table E.3, are modified to handle 128-bit elements. We have not investigated changing any other instructions, but believe it should be trivial to extend them. The RVV specification notes how some instructions would handle 128-bit elements, e.g. [2, Chapter 13].

The RVV memory access instructions are changed to support a new element width (Table E.2). This should be supported in all instructions, but was only tested for unit loads and stores. Indeed, the only instructions added to CHERI-Clang were `vle128.v` and `vse128.v`. Memory access instructions of 128-bit elements are the only instructions that access vector registers in a capability context.

Of the non-unit loads and stores, indexed memory accesses are potentially concerning. These accesses use offsets of the `vtype`-encoded width, so could try to use 128-bit offsets. Indexed memory accesses have not been tested with 128-bit offsets, and further work is required to decide how that case should be handled.

SEW	`vsew[2:0]`		
8	0	0	0
16	0	0	1
32	0	1	0
64	0	1	1
128^{new}	1	0	0

Table E.1: Selected element width encoding

Access Type	`mew`	`width[2:0]`		
Vector(8)	0	0	0	0
Vector(16)	0	1	0	1
Vector(32)	0	1	1	0
Vector(64)	0	1	1	1
Vector$(128)^{new}$	1	0	0	0

Table E.2: Width encoding for vector loads and stores

`vmv.v.v`
`vmv.v.i`
`vmerge.vim`
`vmv<nr>r.v`
`vmseq.vi`
`vmsne.vi`
`vadd.v.i`

Table E.3: `riscv-v-lite` supported arithmetic instructions

E.2.1 Relevant properties

This subsection summarizes the properties of the emulator described in Chapter 5 that enable capabilities-in-vectors. These properties are not absolute requirements for all capability-in-vector implementations fulfilling Hypotheses H-7 to H-9, but can be used as a starting point.

- `ELEN` = 128(+1) i.e. the length of a capability.
  - For a program to manipulate, load, and store capability values securely and atomically, it must be able to operate on appropriately sized logical elements.
- `VLEN` = 128(+1) i.e. the length of a capability.
  - `VLEN` $\geq$ `ELEN` ([2, Chapter 2])
  - Larger `VLEN` could be supported, which must be a power-of-two[2, Chapter 2] and therefore will be a multiple of the capability length.
- The memory interface is identical to that used by the scalar processor, where each vector access is split into a set of sequential accesses less than or equal to 128 bits.
  - Therefore the safety properties for the scalar code still hold.
  - Example: capabilities can only be stored through a capability with the `STORE_CAP` permission.
- Capability memory accesses use `SafeTaggedCap` as a unit.
  - This means it is impossible to set the tag bit on invalid capabilities.
  - It also means all capability accesses are 128-bit aligned, and atomic.

- The only instruction that can place capabilities in a vector register is 128-bit element loads.
  - There are no vectorized capability-to-capability or integer-to-capability instructions, such as `CSetBounds` or `CFromPtr`.
  - All other instructions that write to registers (e.g. arithmetic, 64-bit loads, etc.) unset the tag bit.
  - The only way to place a valid capability in a vector register is to copy a valid capability from memory.
  - Therefore the Provenance and Monotonicity properties are always upheld.
- The only 128-bit element vector load instruction is unit-stride.
  - Strided and Indexed accesses could be supported as long as they enforced alignment and atomicity correctly.
  - Whole-register accesses would need to be updated to always use 128-bit elements, in case capabilities are being accessed.
- Capabilities-in-vectors cannot be dereferenced directly.
  - Therefore Integrity cannot be violated by vector operations.

APPENDIX F

Full test results

F.1 Initial Smoke Tests

Some simple smoke tests were constructed to test the basic functionality of the emulator, particularly under CHERI. `hello_world` runs three small functions which calculate Fibonacci numbers and factorials. Fibonacci is calculated with a simple recursive function, and with *memoization* where previous outputs are cached in a static array. The tests compile on all compilers, and output the correct results on all architectures.

	RV32	RV-64				
	`llvm-13`	`llvm-13`	`llvm-15`	`gcc`	CHERI	CHERI (Int)
`factorial(10)`	Y	Y	Y	Y	Y	Y
`fib(10)` (recursive)	Y	Y	Y	Y	Y	Y
`fib(33)` (memoized)	Y	Y	Y	Y	Y	Y

Table F.1: `hello_world` results — Basic program tests

F.2 `vector_memcpy`

The scope of this test, including testing many permutations of `vtype`, meant the full table couldn't be included in the main paper.

Table F.2: Results — Vectorized memcpy

	RV32	RV-64				
	`llvm-13`	`llvm-13`	`llvm-15`	`gcc`	CHERI	CHERI (Int)
Unit Stride e8m1	Y	Y	Y	Y	Y	Y
Unit Stride e16m2	Y	Y	Y	Y	Y	Y
Unit Stride e32m4	Y	Y	Y	Y	Y	Y
Unit Stride e64m8	Y	Y	Y	Y	Y	Y
Unit Stride e32mf2	Y	Y	Y	-	Y	Y
Unit Stride e16mf4	Y	Y	Y	-	Y	Y
Unit Stride e8mf8	Y	Y	Y	-	Y	Y
Strided e8m1	Y	Y	Y	Y	Y	Y
Strided e16m2	Y	Y	Y	Y	Y	Y
Strided e32m4	Y	Y	Y	Y	Y	Y
Strided e64m8	Y	Y	Y	Y	Y	Y
Strided e32mf2	Y	Y	Y	-	Y	Y
Strided e16mf4	Y	Y	Y	-	Y	Y
Strided e8mf8	Y	Y	Y	-	Y	Y
Indexed e8m1	Y	Y	Y	Y	Y	Y
Indexed e16m2	Y	Y	Y	Y	Y	Y
Indexed e32m4	Y	Y	Y	Y	Y	Y
Indexed e64m8	Y	Y	Y	Y	Y	Y
Indexed e32mf2	Y	Y	Y	-	Y	Y
Indexed e16mf4	Y	Y	Y	-	Y	Y
Indexed e8mf8	Y	Y	Y	-	Y	Y
Unit Stride Masked e8m1	Y	Y	Y	Y	Y	Y
Unit Stride Masked e16m2	Y	Y	Y	Y	Y	Y
Unit Stride Masked e32m4	Y	Y	Y	Y	Y	Y
Unit Stride Masked e64m8	Y	Y	Y	Y	Y	Y
Unit Stride Masked e32mf2	Y	Y	Y	-	Y	Y
Unit Stride Masked e16mf4	Y	Y	Y	-	Y	Y
Unit Stride Masked e8mf8	Y	Y	Y	-	Y	Y
Bytemask Load e8m1	-	-	Y	-	-	-
Bytemask Load e16m2	-	-	Y	-	-	-
Bytemask Load e32m4	-	-	Y	-	-	-
Bytemask Load e64m8	-	-	Y	-	-	-
Bytemask Load e32mf2	-	-	Y	-	-	-

Table F.2: Results — Vectorized memcpy

	RV32	RV-64				
	`llvm-13`	`llvm-13`	`llvm-15`	`gcc`	CHERI	CHERI (Int)
Bytemask Load e16mf4	-	-	Y	-	-	-
Bytemask Load e8mf8	-	-	Y	-	-	-
Unit Stride Segmented e8m2	Y	Y	Y	Y	Y	Y
Unit Stride Segmented e16m2	Y	Y	Y	Y	Y	Y
Unit Stride Segmented e32m2	Y	Y	Y	Y	Y	Y
Unit Stride Segmented e64m2	Y	Y	Y	Y	Y	Y
Unit Stride Segmented e32mf2	Y	Y	Y	-	Y	Y
Whole-Register e64m1	Y	Y	Y	Y	Y	Y
Whole-Register e64m2	Y	Y	Y	Y	Y	Y
Whole-Register e64m4	Y	Y	Y	Y	Y	Y
Whole-Register e64m8	Y	Y	Y	Y	Y	Y
FoF Memcpy e8m1	Y	Y	Y	Y	Y	Y
FoF Memcpy e16m2	Y	Y	Y	Y	Y	Y
FoF Memcpy e32m4	Y	Y	Y	Y	Y	Y
FoF Memcpy e64m8	Y	Y	Y	Y	Y	Y
FoF Memcpy e32mf2	Y	Y	Y	-	Y	Y
FoF Memcpy e16mf4	Y	Y	Y	-	Y	Y
FoF Memcpy e8mf8	Y	Y	Y	-	Y	Y
FoF Boundary e8m1	Y	Y	Y	Y	Y	Y
FoF Boundary e16m2	Y	Y	Y	Y	Y	Y
FoF Boundary e32m4	Y	Y	Y	Y	Y	Y
FoF Boundary e64m8	Y	Y	Y	Y	Y	Y
FoF Boundary e32mf2	Y	Y	Y	-	Y	Y
FoF Boundary e16mf4	Y	Y	Y	-	Y	Y
FoF Boundary e8mf8	Y	Y	Y	-	Y	Y

F.3 `vector_memcpy_pointers`

This is already referenced in the main paper (Section 5.2) and included here for completeness.

	RV32	RV-64				
	`llvm-13`	`llvm-13`	`llvm-15`	`gcc`	CHERI	CHERI (Int)
Copy	Y	Y	Y	Y	Y	Y
Copy + Invalidate	-	-	-	-	Y	Y

APPENDIX G

Artifacts

`riscv-v-lite`

RISC-V ISA emulator and test programs, written from scratch.

Online source code: **(redacted for anonymity)**

Online documentation: **(redacted for anonymity)**

Source code and documentation are also included with the submission.

`rust-cheri-compressed-cap`

Rust wrapper for C capability library, written from scratch.

Online source code: **(redacted for anonymity)**

Online documentation: **(redacted for anonymity)**

Source code and documentation are also included with the submission.

CHERI-Clang fork

LLVM-13-based compiler for CHERI, with changes to make it compatible with RVV.

Online source code: **(redacted for anonymity)**

Source code is also included with the submission.

Online diff, to show our changes: **(redacted for anonymity)**

An offline diff is also included with the submission as `VECTOR.diff`.