Capability-Based Memory Protection for Scalable Vector Processing

Samuel Stark (sws35@cam.ac.uk)

June 16th 2022

Impenetrable Title

Capability-Based Memory Protection for Scalable Vector Processing

Impenetrable Title

Capability-Based Memory Protection for Scalable Vector Processing

Impenetrable Title

Capability-Based Memory Protection for Scalable Vector Processing

Background - What is CHERI?

Capability-Based Memory Protection == CHERI[1]

- Memory is normally addressed with integers
    - Integer addresses can be *forged*
    - Code can be tricked into accessing memory it shouldn't
- CHERI architectures use *capability* addressing
    - Capability = Bounds + current address
- Capabilities cannot be forged, only *derived* from other capabilities

Code can only access memory when it has been *given* access to that memory

[1]Capability Hardware Enhanced RISC Instructions[1]

Background - What are vectors?

(Scalable) Vector Processing

- Vector architectures allow programmers to use SIMD
- Most vector architectures have *fixed-length* vectors
  - SSE, Arm Neon = 128-bit, AVX-512 = 512
  - Vector lengths have hardware tradeoff
  - Need to recompile code for different vector lengths
- *Scalable vector* architectures give designers flexibility[2]
  - Code doesn't rely on fixed vector length

24	9	9	28

+

13	17	10	24

=

37	26	19	52

Table 1: Vector addition

Have we combined them before?

- CHERI affects vector memory accesses
  - Loading N vector elements in a single instruction
  - Per-element bounds checks could be expensive?
- Arm have manufactured CHERI hardware
  - Has fixed-length SIMD
  - Doesn’t support complex access patterns
- No other general-purpose CHERI processors with vector support

Where does this matter?

Background - Where does it matter most?

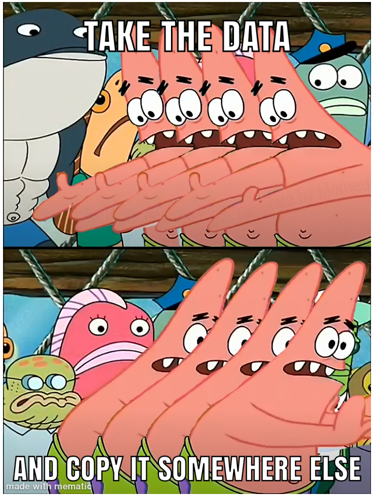
`memcpy`!

- Take data and copy it somewhere else
- Extremely widespread operation
- Vectors can copy more data per instruction
- Vectorized **`memcpy`** should work and go fast on CHERI

Background - Where does it matter most?

Vectorized memcpy!

- Take data and copy it somewhere else
- Extremely widespread operation
- Vectors can copy more data per instruction
- Vectorized **`memcpy`** should work and go fast on CHERI



Project goal

Make vectorized `memcpy` functional and fast on CHERI

Combine the RISC-V Vector extension (RVV) with CHERI-RISC-V

1. Write a RISC-V CHERI-RVV emulator in Rust
   - Demonstrates hardware feasibility
2. Write test programs in C
   - Demonstrates software feasibility
3. Run the test programs on the emulator!

Project goal

Make vectorized `memcpy` functional and fast on CHERI

Combine the RISC-V Vector extension (RVV) with CHERI-RISC-V

RVV (original)

- Uses integer addressing
- Loads/stores integer data

CHERI-RVV (ours)

- Uses capability addressing
  - Performance concerns?
- Loads/stores integers and capabilities
- Doesn’t break CHERI security

Step 1: Making vector accesses *use* capabilities

Vector memory access patterns

1. Unit-stride
   - `base, base+1, base+2...`
2. Strided
   - `base, base+n, base+2n...`
3. Indexed
   - `base + index[0], base + index[1], base + index[2]...`

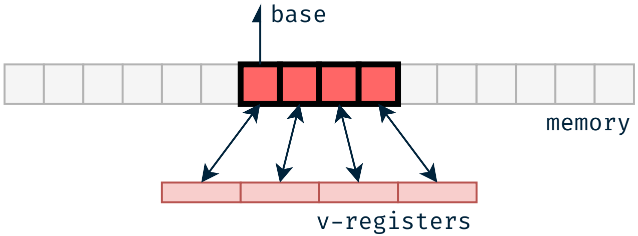


Figure 1: Unit access

Vector memory access patterns

1. Unit-stride
    - `base, base+1, base+2...`
2. Strided
    - `base, base+n, base+2n...`
3. Indexed
    - `base + index[0], base + index[1], base + index[2]...`

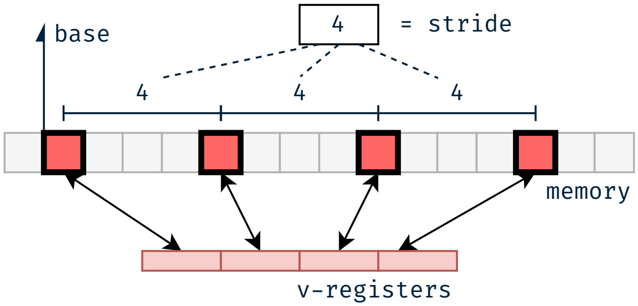


Figure 2: Strided

Vector memory access patterns

1. Unit-stride
   - `base, base+1, base+2...`

2. Strided
   - `base, base+n, base+2n...`

3. Indexed
   - `base + index[0], base + index[1], base + index[2]...`

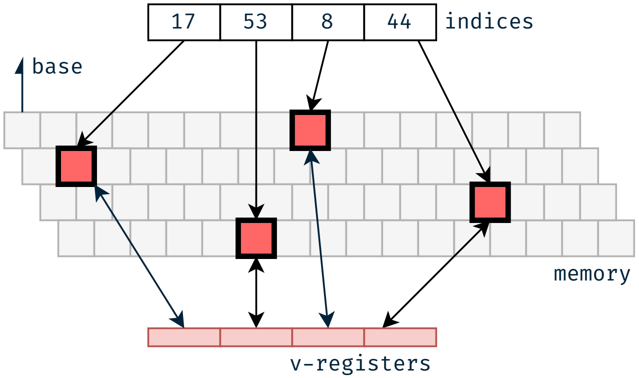
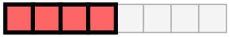
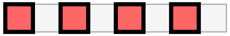
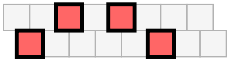
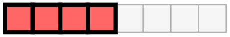
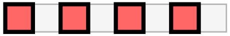
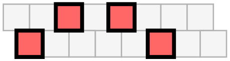


Figure 3: Indexed

Vector memory access patterns

Property	Example value	Memory pattern
Unit-stride		
Base address	`0x37f0`	
Strided		
Base address Stride	`0x37f0` 2	
Indexed		
Base address Index vector	`0x37f0` 8 2 13 4	

Vector memory access patterns

Property	Example value	Memory pattern
Unit-stride		
Base capability	`0x3700.. 0x37f0 ..0x3800`	
Strided		
Base capability Stride	`0x3700.. 0x37f0 ..0x3800` 2	
Indexed		
Base capability Index vector	`0x3700.. 0x37f0 ..0x3800` 8 | 2 | 13 | 4	

Step 2: Making vector accesses *copy* capabilities

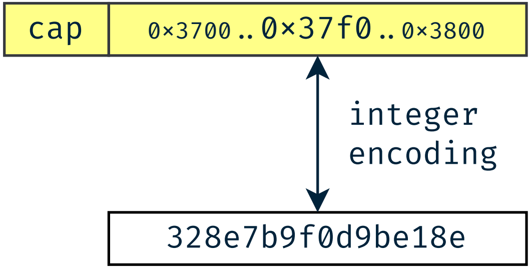
Storing capabilities in memory

- Memory can hold both capabilities and integers
- Separate *tag memory* denotes which regions are capabilities
    - Access to tag memory is controlled by hardware
- Without the tag, you get the *integer encoding* of the capability

tag	data[128:0]
int	4436c97773d3504f
cap	0x3700 .. 0x37f0 .. 0x3800
	...
int	

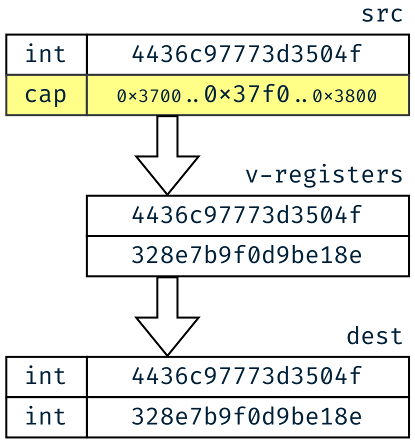
Storing capabilities in memory

- Memory can hold both capabilities and integers
- Separate *tag memory* denotes which regions are capabilities
    - Access to tag memory is controlled by hardware
- Without the tag, you get the *integer encoding* of the capability



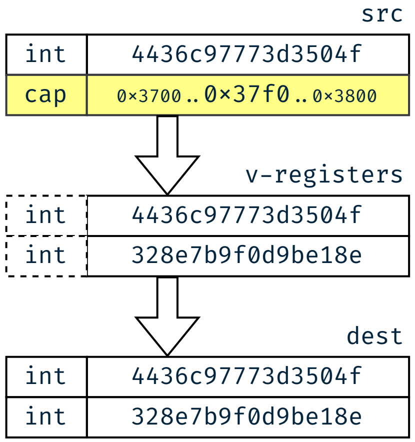
Integer-only memcpy

- The original RVV specification doesn't consider capabilities
- Assumes vectors only hold integer data
- ⇒ `memcpy` converts capabilities to integers :(



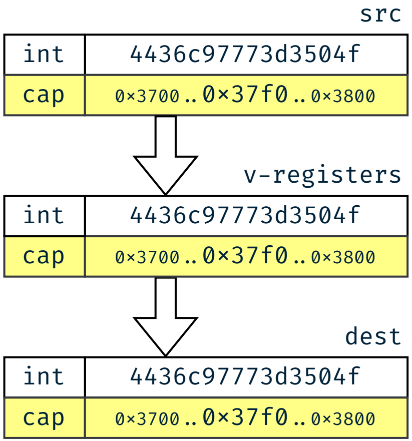
Integer-only memcpy

- The original RVV specification doesn't consider capabilities
- Assumes vectors only hold integer data
- ⇒ `memcpy` converts capabilities to integers :(



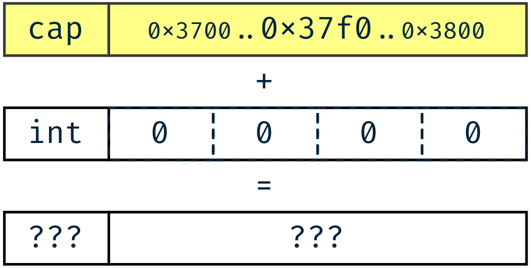
Copying capabilities in vectors

- If we add tag bits to vector registers, we can load/store them correctly
- But does that make anything else more complicated?
  - Yes



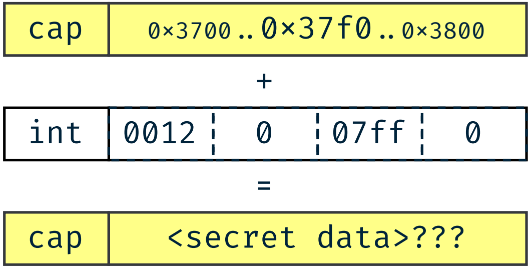
Storing capabilities in vectors???

- Now *all vector instructions* can interact with capabilities!
- If we aren't careful, attackers could forge capabilities
- We introduce two contexts of accessing vector registers
    - Integer context
    - Capability context



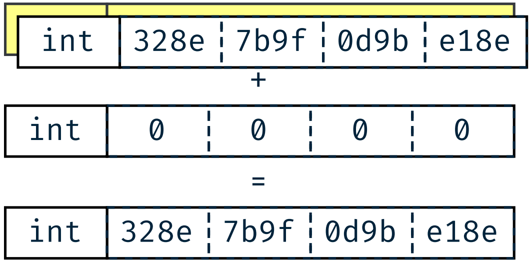
Storing capabilities in vectors???

- Now *all vector instructions* can interact with capabilities!
- If we aren't careful, attackers could forge capabilities
- We introduce two contexts of accessing vector registers
    - Integer context
    - Capability context

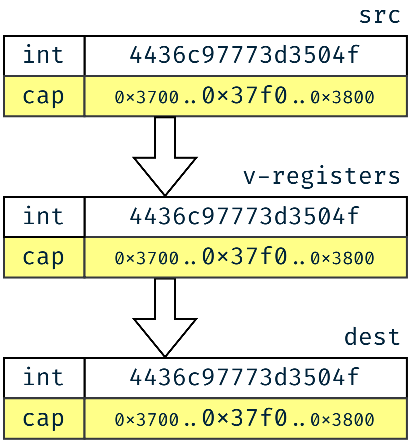


Storing capabilities in vectors???

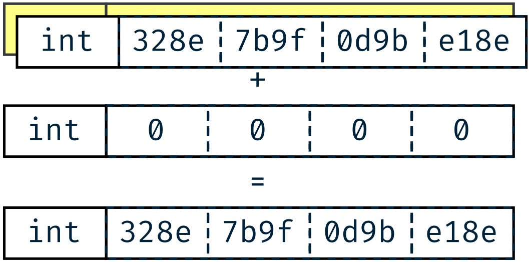
- Now *all vector instructions* can interact with capabilities!
- If we aren't careful, attackers could forge capabilities
- We introduce two contexts of accessing vector registers
  - Integer context
  - Capability context



Integer/Capability context



Capability context
(128-bit vector loads/stores)



Integer context
(Everything else)

memcpy works!

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

Conclusion

CHERI-RVV Summary

Uses capability addressing
Loads/stores integers and capabilities
Doesn’t break CHERI security

Supports all vanilla RVV instructions
Is binary-compatible with vanilla RVV
Can be* source-compatible with vanilla RVV

Has a reference implementation:
Emulator, compiler*, test programs
9,500 LoC

*compiler requires engineering work

Future work

Do more with vectors than just

1. Vectorized `memcpy`
2. Vectorized tag clearing

Add new instructions for e.g. temporal revocation[3]?

Samuel Stark
sws35@cam.ac.uk

Conclusion

CHERI-RVV Summary

Uses capability addressing
Loads/stores integers and capabilities
Doesn’t break CHERI security

Supports all vanilla RVV instructions
Is binary-compatible with vanilla RVV
Can be* source-compatible with vanilla RVV

Has a reference implementation:
Emulator, compiler*, test programs
9,500 LoC

*compiler requires engineering work

Future work

Do more with vectors than just

1. Vectorized `memcpy`
2. Vectorized tag clearing

Add new instructions for e.g. temporal revocation[3]?

Samuel Stark
sws35@cam.ac.uk

Per-element checks

- Vector hardware can coalesce element accesses
  - e.g. 4x 32-bit elements in the same cache line can be transferred over a 128-bit bus at once
- Want to coalesce the per-element capability checks as well
  - Otherwise they could bottleneck
  - Or use too much logic
- We found we *can* coalesce capability checks if they succeed
  - i.e. ”is the cache line inside the capability bounds”
  - But if that check fails, we have to check each element individually
  - RVV requires that any synchronous exception (i.e. capability check) reports the element that triggered it

References

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

[2] 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`.

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