Securing Mixed Rust with Hardware Capabilities

Muhammad Hassnain

Literature Review

Sep 12, 2025

Disclaimer!

- I do not really understand this paper very much, so this might be a half baked discussion than a presentation.
- This paper builds on top on top of Capstone (USENIX'23), I didn't have enough time to read the other paper :(

Capabilities

- A capability c contains a pair of physical addresses encoding the starting and ending addresses of the permitted access regions.
- The permission p encodes the permitted access operation on the capability, which be RW , RO, an no-access.
- The hardware distinguishes capabilities from raw data and tracks.

Rust Principles

```
1  char *buf = malloc(4096);
2  // ...
3  free(buf);
4  // ...
5  memcpy(buf, inp, 4096);
```

What's wrong in this C Code?

Rust Principles

```c
char *buf = malloc(4096);
// ...
free(buf);
// ...
memcpy(buf, inp, 4096);
```

Problematic C code

```rust
let mut buf = vec![0u8; 4096];
// ...
drop(buf);
// ...
buf.copy_from_slice(&inp);
```

Rust compiler rejects this

```
error[E0382]: borrow of moved value: `buf`
 --> src/main.rs:5:5
```

Rust Principles

```
1  char buf1[16];
2  // ...
3  char *buf2 = buf1 + 2;
4  memcpy(buf2, buf1, 8);
5  if (memcmp(buf2, key, 8) == 0)
6    seteuid(0);
```

What's wrong in this C Code?

Rust Principles

```c
char buf1[16];
// ...
char *buf2 = buf1 + 2;
memcpy(buf2, buf1, 8);
if (memcmp(buf2, key, 8) == 0)
  seteuid(0);
```

What's wrong in this C Code?

```rust
let mut buf1 = [0u8; 16];
// ...
let buf2 = &mut buf1[2..10];
buf2.copy_from_slice(
  &buf1[..8]
);
if *buf2 == *key {
  seteuid(0);
}
```

Rust compiler rejects this

```
error[E0502]: cannot borrow `buf1` as immutable because it is also borrowed as mutable
 --> src/main.rs:4:27
```

Rust Principles

What's wrong in this C Code?

```c
void consumer(void *buf) {
  // ...
  ((int*)buf)[10] = 1;
  // ...
}
int main(void) {
  int *buf = malloc(sizeof(int)
  ↪  * 16);
  // ...
  pthread_create(&tc, NULL,
  ↪  &consumer, buf);
  // ...
  buf[10] = 0;
  free(buf);
}
```

Rust Principles

```c
void consumer(void *buf) {
  // ...
  ((int*)buf)[10] = 1;
  // ...
}
int main(void) {
  int *buf = malloc(sizeof(int)
  ↪ * 16);
  // ...
  pthread_create(&tc, NULL,
  ↪ &consumer, buf);
  // ...
  buf[10] = 0;
  free(buf);
}
```

What's wrong in this C Code?

```rust
fn consumer(buf : &mut [i32]) {
  // ...
  buf[10] = 1;
  // ...
}
fn main() {
  let mut buf = vec![0i32; 16];
  // ...
  std::thread::spawn(|| {
    consumer(&mut buf[..]);
  });
  // ...
  buf[10] = 0;
}
```

```
error[E0499]: cannot borrow `buf` as mutable more than once at a time
  --> src/main.rs:14:5
```

Rust Principles (Recap)

- Ownership
- Borrowing
- Aliasing XOR Mutability

Q : How do you guarantee these are upheld after FFI calls and inline assembly ?

Rust Principles Violation

Is this correct?

```rust
fn free_xor_alias(raw: *mut u8, obj: Box<u8>) {
    assert_eq!(*obj, 0);
    unsafe { *raw = 42; }
    if *obj != 42 {
        unsafe { libc::free(raw as *mut libc::c_void); }
    }
}

fn main() {
    let mut obj = Box::new(0);
    free_xor_alias(std::ptr::addr_of_mut!(*obj.as_mut()), obj);
}
```

Rust Principles Violation

If compiler optimizations are enabled , the compiler assumes raw and obj are not aliased, therefore obj is unmodified since line2 check and hence it evaluates line 4 as always true.

```
1  fn free_xor_alias(raw: *mut u8, obj: Box<u8>) {
2      assert_eq!(*obj, 0);
3      unsafe { *raw = 42; }
4      if *obj != 42 {
5          unsafe { libc::free(raw as *mut libc::c_void); }
6      }
7  }
8
9  fn main() {
10     let mut obj = Box::new(0);
11     free_xor_alias(std::ptr::addr_of_mut!(*obj.as_mut()), obj);
12 }
```

Types of Capabilities

- Linear Capability

  “Linear capabilities are not duplicable. Capstone guarantees regions pointed to by linear capabilities do not overlap with other capabilities of the system”

Types of Capabilities

- Linear Capability

  “Linear capabilities are not duplicable. Capstone guarantees regions pointed to by linear capabilities do not overlap with other capabilities of the system”

- Revocation Capability

  “Enables a selective revocation mechanism”

Hmmm…

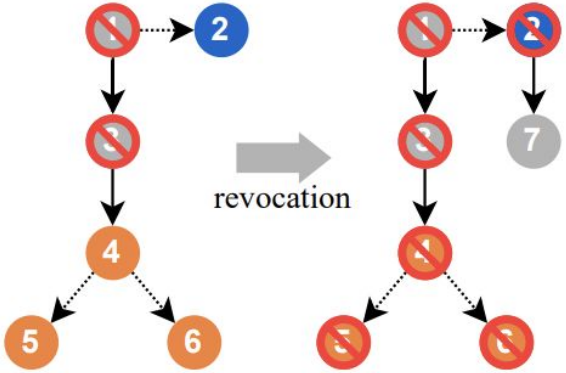


Figure 2: Overview of the revocation mechanism in Capstone (baseline). Each circle represents a capability, with the number indicating the order in which they are created. The colour indicates the capability type: linear (gray), non-linear (orange), and revocation (blue). Solid arrows represent capability derivations that destroy the original capabilities, and dashed arrows represent derivations that retain them.

Does this work on Rust code?

Ignore the numbers for now.

Can you spot the problem here?

```
1  use std::arch::asm;
2  fn use_p(p 2 : *mut u64) {
3    unsafe {
4      asm!(
5        "sd x0, 0({addr})",
6        addr = in(reg) p 2
7      );
8    }
9  }
10 fn main() {
11   let mut v 1 = Box::new(0u64);
12   let v_raw 2 = &mut *v as *mut u64;
13   let v_ref 3 = unsafe { &mut *v_raw 2 };
14   use_p(v_raw 2);
15   *v_ref 3 = 42;
16 }
```

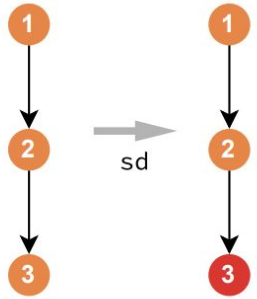
Capstone : Revoke on Use Model!

- Perform revocation implicitly when software uses a capability to access a memory.
- Allows borrow operation to build capability derivation.
- A freshly borrowed capability is always unique, and implicit revocation ensures, while a borrow lives, there is no access to that memory through any other capability.

Revisit this again!

Using 2 invalidates 3.

```
1  use std::arch::asm;
2  fn use_p(p 2 : *mut u64) {
3    unsafe {
4      asm!(
5        "sd x0, 0({addr})",
6        addr = in(reg) p 2
7      );
8    }
9  }
10 fn main() {
11   let mut v 1 = Box::new(0u64);
12   let v_raw 2 = &mut *v as *mut u64;
13   let v_ref 3 = unsafe { &mut *v_raw 2 };
14   use_p(v_raw 2);
15   *v_ref 3 = 42;
16 }
```



(a) (b)

Figure 3: Example of CapsLock detecting a Rust principle violation. Circles represent capabilities. Those in red are invalid. (a) Code with the capability associated with each value indicated. (b) CapsLock maintains capability derivations in trees and invalidates capabilities based on them.

Operations on Revoke on Use Capabilities

Copy : When copied through a mv instruction, treats both as identical and allow them to be used interchangeably.

Borrow: Temporarily transfers exclusive access to the new one. Also support partial borrowing.

Creation: Root of borrow tree, newly allocated stack or heap objects.

Revocation: Used when software frees the memory.

Load/Store Checked with corresponding memory load/store

Security Invariants

- Well Nested Borrowing:

  A capability cannot outlive the capability it borrows from.

Security Invariants

- Well Nested Borrowing:

  A capability cannot outlive the capability it borrows from.

- Exclusive Access
  - Consider capability c, and D(c) is set of all capabilities derived from it (including c)
  - During the lifetime of c, there is no store to an aliasing memory location from outside D(c), and similarly for load as well.
  - This is basically the Rust mut/immut constraint.

Comparison to Rust Memory Model

- More relaxed than both stack and tree borrow.
- Both stack and tree borrow invalidate pointers upon borroiwng, Capslock does it more lazily.
- CapsLock allows all mutable references to be reserved and treated as immutable until the first mutable.
- Does not enforce protectors for function arguments.

Capslock Instruction Semantics

Fancy Looking table 👀

Set representation of capabilities?

Instruction	Rule
Create (a_l, a_r)	$a_l < a_r \quad i \notin \mathrm{dom}(m)$ $m' = m[i \mapsto (a_l, a_r, \mathrm{RW}, \bot)]$
BorrowR (i, a'_l, a'_r)	$m(i) = (a_l, a_r, p, b) \quad p \neq \mathrm{NA} \quad i' \notin \mathrm{dom}(m)$ $a_l \leq a'_l < a'_r \leq a_r \quad m' = m[i' \mapsto (a'_l, a'_r, \mathrm{RO}, i)]$
BorrowW (i, a'_l, a'_r)	$m(i) = (a_l, a_r, p, b) \quad p = \mathrm{RW} \quad i' \notin \mathrm{dom}(m)$ $a_l \leq a'_l < a'_r \leq a_r \quad m' = m[i' \mapsto (a'_l, a'_r, \mathrm{RW}, i)]$
Drop (i)	$m(i) = (a_l, a_r, p, b) \quad m' = \mathrm{RevokeW}(m, S)$ $p \neq \mathrm{NA} \quad S = D_m(R_m(i)) \cup \{R_m(i)\}$
Load (i, a)	$m(i) = (a_l, a_r, p, b) \quad a_l \leq a < a_r \quad p \neq \mathrm{NA}$ $m' = \mathrm{RevokeR}(m, D_m(S)) \quad S = O_m(a) \cap \overline{A_m(i)}$
Store (i, a)	$m(i) = (a_l, a_r, p, b) \quad a_l \leq a < a_r \quad p = \mathrm{RW}$ $m' = \mathrm{RevokeW}(m, D_m(S)) \quad S = O_m(a) \cap \overline{A_m(i)}$

Implementation

- Capability based hardware , which adds support for revoke on using capabilities and related instructions using QEMU.
- Some instrumentation to Rust code to expose key semantics to the hardware.

Rust Specific Adjustments

- Raw Pointers
  - Softwares can use aliasing raw pointers created from the same reference in an interleaving way.
  - While a raw pointer type capability is valid, no store to any memory location, that aliases with the capability memory bounds takes places unless through a capability derived from its parent though borrowing.

Rust Specific Adjustments

- Interior Mutability
  - Rust can allow mutation even behind immutable references, if it is done behind an Unsafe Cell reference.
  - They relax exclusiveness invariant to except aliasing stores within subtress of Unsafe Cell capabilities ,while a capability is valid, no sore to any memory location that aliases with the capability memory takes place unless through a capability derived from it through borrowing (inside its subtree) or derived form an UnsafeCell.

Rust Specific Adjustments

- Pointer Provenance
  - Sometimes some things can span two registers…
  - So they allow multiple capabilities to be linked to one object.
  - Also useful for deriving the capability of the result.
  - They resolve the result capability when it's first load/store based on the corresponding memory address and capability to that.

Rust Specific Adjustments

- They also instrument compiler to add another MIR pass, to inject these capability instructions.
- Their MIR pass, checks for conversions b/w references and raw pointers and injects borrow instructions immediately after each conversion.
- They also modify prologue and epilogue to cover stack objects.

Evaluation

- Is Capslock compatible with existing Rust code?
- Can CapsLock detect new bugs in mixed Rust code?
- What is the expected performance overhead?

Evaluation

Can CapsLock detect new bugs in mixed Rust code? (just focusing on this)

- Detects 15/15 Rustsec examples (same performance as MIRI).

  (I wanted to include those exploits in the presentation :( )

- For Mixed code, they pick 1000 popular crates, along with 13000 randomly sampled crates deemed to use FFI. (detects 8 unknown cases)

Evaluation

Simplified example of AXM violation in generator.

```
1  let mut r = Box::new(N::new());
2  let p = &mut *r as *mut _;
3  r.p = p;
4  // ...
5  unsafe { &mut *r.p }
```

Evaluation

The C function uses the raw pointer started.compress.cinfo to mutate the contents of started.dest_mgr.

Link

Playground

```
let mut started = CompressStarted {
  compress: self,
  dest_mgr: Box::new(DestinationMgr::new(writer,
  ↪  write_buffer_capacity)),
};
unsafe {
  let dest_ptr = addr_of_mut!(started.dest_mgr.as_mut().iface);
  started.compress.cinfo.dest = dest_ptr;
  ffi::jpeg_start_compress(&mut started.compress.cinfo,
  ↪  boolean::from(true));
}
// ...
unsafe {
  ffi::jpeg_finish_compress(&mut started.compress.cinfo);
}
```

Evaluation

They are 2x faster than MIRI on average.

They are compatible with 99.7% of test cases in top 100 crates.