Parsing binary formats with zerocopy

2025-02-05

Florian “Florob” Zeitz

Outline

1 Introduction

Parsing

- create a data structure fit for programmatic processing
- break data up into logical parts
- handle endianness

Regular Approach

```rust
#[derive(Debug, serde::Deserialize)]
struct Stuff {
    _x: u16,
    _s: String,
}

fn main() {
    let data = b"\x2a\x00\x0d\x00\x00\x00\x00\x00\x00\x00Hello Cologne";
    let stuff: Stuff = bincode::deserialize(data).unwrap();
    println!("{stuff:?}");
}
```

Don’t Copy Large Amounts of Data

```rust
#[derive(Debug, serde::Deserialize)]
struct Stuff<'a> {
    _x: u16,
    _s: &'a str,
}

fn main() {
    let data = b"\x2a\x00\x0d\x00\x00\x00\x00\x00\x00\x00Hello Cologne";
    let stuff: Stuff = bincode::deserialize(data).unwrap();
    println!("{stuff:?}");
}
```

2 The C Approach (and Its Problems)

Zero-copy Parsing by Type Punning

- a buffer pointer is cast to a `struct` pointer mimicking the binary format
- fields of the `struct` can be read to “parse” the data in place
- writing to fields also allows modifying the data in place

```
1 char buffer[20] = {0x40, 0};
2 struct iphdr *ip = (struct iphdr *)buffer;
3 assert(ip→version == 4);
```

IPv4 Struct (adapted from linux/ip.h)

```
 1 struct iphdr {
 2     uint8_t version:4, ihl:4;
 3     uint8_t tos;
 4     uint16_t total_length;
 5     uint16_t id;
 6     uint16_t fragment_offset;
 7     uint8_t ttl;
 8     uint8_t protocol;
 9     uint16_t header_checksum;
10     uint32_t saddr;
11     uint32_t daddr;
12     /* The options start here. */
13 };
```

- can be used to parse an IPv4 header
- packing of bit-fields is implementation defined
- order of bit-fields is implementation defined (endianness dependent in gcc)
- fields may not be stored in platform-endianness
  - remember to use `ntohs()`/`ntohl()`

Strict Aliasing

C23

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- a type compatible with the effective type of the object,
- a qualified version […],
- the signed or unsigned type […],
- […]
- a character type.

- accessing a character type as another type is not (explicitly) legal
- some argue only `char` is legal, not `unsigned char` or `uint8_t`
- can be disabled (e.g. Linux does), but is important for performance

Padding

Alignment

Objects have to occur at an address divisible by their types alignment. The alignment is typically their size, but can be larger or smaller. Structs are aligned according to their members.

```
1  struct ethhdr {
2    uint8_t dest_mac[6];
3    uint8_t src_mac[6];
4    uint16_t ethertype;
5  };
6
7  struct frame {
8    struct ethhdr eth;
9    /* 2 bytes padding */
10   struct iphdr ip;
11 };
```

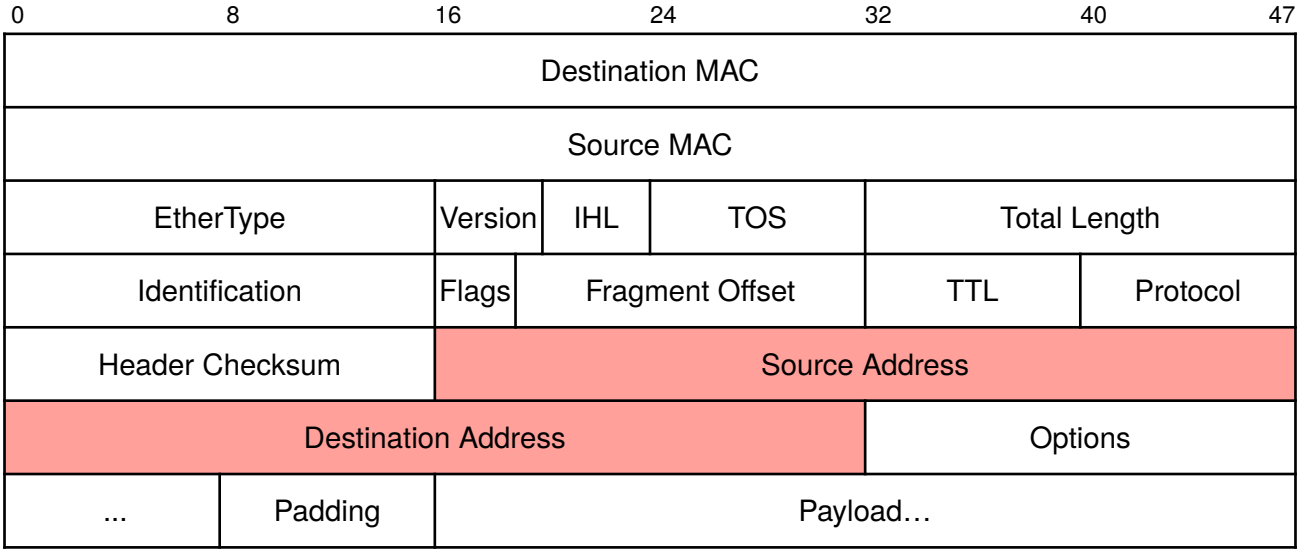
- `struct iphdr` has alignment 4 as it contains multiple `uint32_t`
- padding is inserted due to alignment requirements
- padding does not exist in the original frame
- wrong results when used to parse data

Packed

```
1  struct ethhdr {
2    uint8_t dest_mac[6];
3    uint8_t src_mac[6];
4    uint16_t ethertype;
5  };
6
7  struct frame {
8    struct ethhdr eth;
9    struct iphdr ip;
10 } __attribute__((packed));
```

- specify no padding should be inserted between fields
- not standard C
- causes fields to be misaligned
- creating a pointer to a misaligned field is UB

Misaligned


0
8
16
24
32
40
47
Destination MAC
Source MAC
EtherType
Version
IHL
TOS
Total Length
Identification
Flags
Fragment Offset
TTL
Protocol
Header Checksum
Source Address
Destination Address
Options
...
Padding
Payload…

3 Rust Semantics

Struct Layout

```
1  struct Iphdr {
2      version_ihl: u8,       // 0x10
3      tos: u8,               // 0x11
4      total_length: u16,     // 0x08
5      id: u16,               // 0x0A
6      fragment_offset: u16, // 0x0C
7      ttl: u8,               // 0x12
8      protocol: u8,          // 0x13
9      header_checksum: u16, // 0x0E
10     saddr: u32,            // 0x00
11     daddr: u32,            // 0x04
12 }
```

- field order is not guaranteed
- generally reordered to ensure minimal size
- can be overridden with `#[repr(C)]` or `#[repr(packed)]`
- `#[repr(packed)]` is part of the language
- taking a reference to an unaligned field is a compile time error
- no bit-fields

Aliasing

- Rust has much **stricter** aliasing guarantees
  - when aliases exist they are all read-only
  - objects can be mutated only when no aliases exist
- Rust does not (and will never) have type-based aliasing restrictions
  - type-punning is legal
  - value still has to be valid for the type

Parsing by Transmuting Values

```rust
use core::mem;

let mut buffer = [0u8; mem::size_of::<Iphdr>()];
buffer[0] = 0x40;
let ip: Iphdr = unsafe { mem::transmute(buffer) };
assert_eq!(ip.version_ihl >> 4, 4);
```

- transmute the actual data to a different type
- exclusive to Rust
- `unsafe`: original data might be invalid for new type
- only works if the buffer has exactly the right length
- fields won't have platform-endianness

Parsing by Transmuting References

```
1 let mut buffer = [0u8; 42];
2 buffer[0] = 0x40;
3 let ip: &Iphdr = unsafe { mem::transmute(&buffer) };
4 assert_eq!(ip.version_ihl >> 4, 4);
```

- transmute a reference to the data
- `unsafe`: invalid data or too little data
- works this easily only for references to arrays, not slices
- fields won't have platform-endianness

4 zerocopy crate

zerocopy

- **zerocopy** realizes a concept called “safe transmutation”
- arbitrary objects of the same size can be converted to each other, if:
  - values can be safely constructed from arbitrary bytes (e.g. `u32`)
  - values can be safely constructed from a certain byte sequence after runtime checks (e.g. `bool`)
- supports conversion from slices, including just a prefix/suffix
- includes endianness-aware integer types

Conversion Traits

Conversion is realized with a set of traits. These **must** be derived.

FromBytes

indicates that a type may safely be converted from an arbitrary byte sequence

IntoBytes

indicates that a type may safely be converted to a byte sequence

TryFromBytes

indicates that a type may safely be converted from certain byte sequences (conditional on runtime checks)

Marker Traits

Markers traits are required to call certain functions. These **must** be derived. Derive all you can.

KnownLayout

indicates that zerocopy can reason about certain layout qualities of a type

Immutable

indicates that a type is free from interior mutability,
except by ownership or an exclusive (`&mut`) borrow

Unaligned

indicates that a type's alignment requirement is 1

Endianness-Aware Types

```rust
use zerocopy::byteorder::{U32, BE};

let mut x = U32::<BE>::from_bytes([0, 0, 0, 0x2a]);
assert_eq!(x.get(), 42);

x.set(23);
assert_eq!(x.as_bytes(), &[0, 0, 0, 0x17]);
```

- endianness-aware abstraction
- alignment of one (avoids accidental padding)
- type aliases `zerocopy::byteorder::big_endian::U32`

zerocopy IP Header

```rust
use zerocopy::byteorder::big_endian::{U16, U32};
use zerocopy_derive::{FromBytes, Immutable, IntoBytes, KnownLayout, Unaligned};

#[repr(C)]
#[derive(FromBytes, IntoBytes, KnownLayout, Unaligned, Immutable)]
struct Iphdr {
    version_ihl: u8,
    tos: u8,
    total_length: U16,
    id: U16,
    fragment_offset: U16,
    ttl: u8,
    protocol: u8,
    header_checksum: U16,
    saddr: U32,
    daddr: U32,
}
```

Usage: Safe Transmute

```
let mut buffer = [0u8; 20];
buffer[0] = 0x40;
let ip: Iphdr = zerocopy::transmute!(buffer);
assert_eq!(ip.version_ihl >> 4, 4);
```

Usage: Via FromBytes

```
let mut buffer = [0u8; 42];
buffer[0] = 0×40;
let (ip, _rest) = Iphdr::ref_from_prefix(&buffer).unwrap();
assert_eq!(ip.version_ihl >> 4, 4);
```

Ref

```rust
fn iphdr_from_bytes<B: zerocopy::SplitByteSlice>(
    buffer: B,
) -> Result<zerocopy::Ref<B, Iphdr>, zerocopy::CastError<B, Iphdr>> {
    zerocopy::Ref::from_prefix(buffer).map(|(hdr, _rest)| hdr)
}

let mut buffer = [0u8; 42];
let mut ip = iphdr_from_bytes(&mut buffer[..]).unwrap();
ip.version_ihl = 0x40;
ip.saddr.set(0xc0_a8_00_04);
```

- reference type that acts as `&T` or `&mut T` depending on the underlying buffer
- can reference the whole buffer, or just a slice

Custom DSTs

```
#[repr(C)]
#[derive(FromBytes, KnownLayout, Unaligned, Immutable)]
struct Ip {
    hdr: Iphdr,
    payload: [u8],
}

let buffer = [0u8; 42];
let ip = Ip::ref_from_bytes(&buffer).unwrap();
assert_eq!(ip.payload.len(), 22);
```

Questions?

https://babelmonkeys.de/~florob/talks/RC-2025-02-05-zerocopy-parsing.pdf

Thank you for your attention. Any questions?