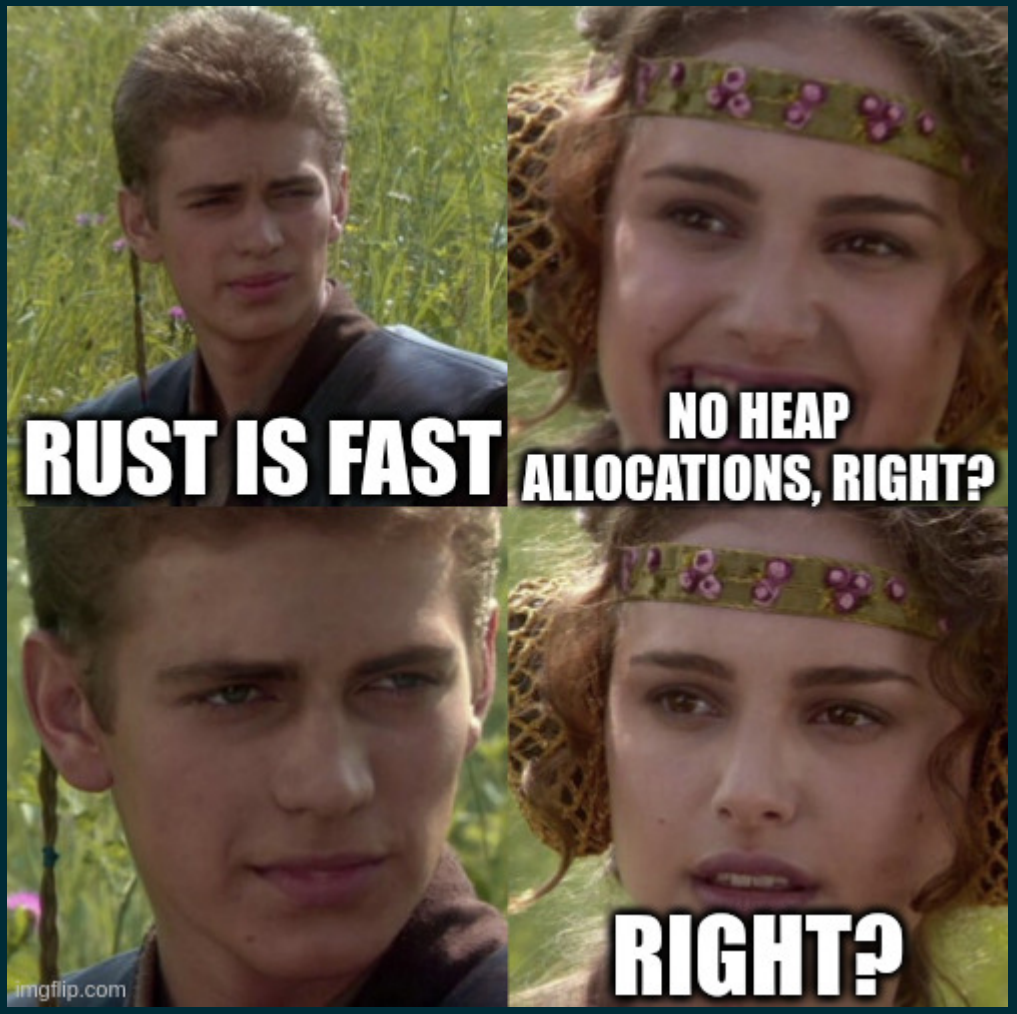
HEAP ALLOCATIONS



```
let mut state = AppState::new();
let mut tun = Tun::new(...);
let mut udp = UdpSocket::bind(...);

loop {
    let ip_packet = tun.next_packet().await;

    let udp_packet = state.encrypt(ip_packet);

    udp.send(udp_packet).await;
}
```

```
let mut state = AppState::new();
let mut tun = Tun::new(...);
let mut udp = UdpSocket::bind(...);

loop {
    let ip_packet = tun.next_packet().await;

    let udp_packet = state.encrypt(ip_packet);

    udp.send(udp_packet).await;
}
```

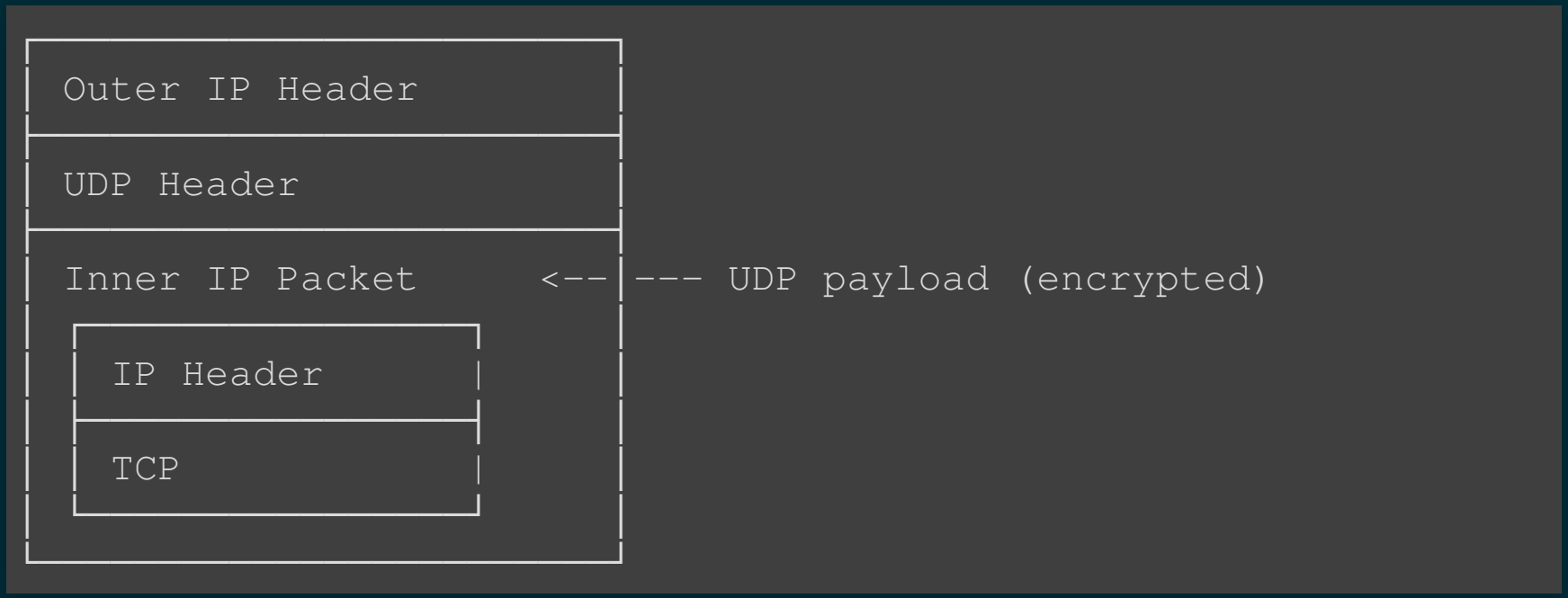
```
let mut state = AppState::new();
let mut tun = Tun::new(...);
let mut udp = UdpSocket::bind(...);

loop {
    let udp_packet = udp.next_packet().await;

    let ip_packet = state.decrypt(udp_packet);

    tun.send(ip_packet).await;
}
```

PACKET STRUCTURE



BUFFERS & LIFETIMES

```
let mut buf = vec![0u8; u16::MAX] // lifetime 'a starts here.

loop {
    let p: Packet<'a> = udp.next_packet(&'a mut buf).await;
}
```

SINGLE LOOP

```
let mut state = AppState::new();

let mut tun = Tun::new(...);
let mut tun_buffer = vec![0u8; u16::MAX]

let mut udp = UdpSocket::bind(...);
let mut udp_buffer = vec![0u8; u16::MAX]

loop {
    let packet = select(
        udp.next_packet(&mut udp_buffer),
        tun.next_packet(&mut tun_buffer)
    ).await;

    let packet = state.handle(packet);

    match packet {
        Packet::Udp(p) => udp.send(p).await,
        Packet::Ip(p) => tun.send(p).await
    }
}
```

SANS-IO

- `AppState` is sans-io: Doesn't know anything about sockets
- Easily unit-testable
- More here: https://www.firezone.dev/blog/sans-io

PERF CHARACTERISTICS

- Crypto: ChaCha20 + Poly1305
- `tokio::net::UdpSocket`
- TUN device is `tokio::io::unix::AsyncFd` of `/dev/net/tun`

PERF CHARACTERISTICS

- Crypto: ChaCha20 + Poly1305
- `tokio::net::UdpSocket`
- TUN device is `tokio::io::unix::AsyncFd` of `/dev/net/tun`

Component	CPU-time
Crypto	12%
UDP	33%
TUN	31%

DESIGN ANALYSIS

- Single threaded
- Pass references back and forth
- No mutexes
- Easy to reason about
- Easy to test

WHAT CAN WE OPTIMISE?

1. Do things in parallel:
    - Can only encrypt one packet at a time: `AppState` is the theoretical bottleneck
    - IP & UDP packets are independent of each other
2. Make syscalls "cheaper", i.e. send more than 1 packet in a single syscall
    - GRO
    - GSO

THE PROBLEM WITH PARALLELISATION

- References cannot be sent across threads
- We only have a single buffer

PACKET SIZES

- Theoretical size of an IP packet: **65 Kb**
- Practical size of an IP packet: **~1500 bytes**
- TUN MTU is limited to 1280 bytes

Adapter	MTU
loopback	65536
wlp1s0	1500
eth0	1500
tun-firezone	1280

ALLOCATE ON THE STACK

- 1280 bytes is a not small but easily fits on the stack
- Allocate each packet on the stack
- No more references -> can be sent across threads
- **But:** `memcpy`

Src: https://github.com/firezone/firezone/pull/6673

ALLOCATE ON THE STACK

- 1280 bytes is a not small but easily fits on the stack
- Allocate each packet on the stack
- No more references -> can be sent across threads
- **But:** `memcpy`
- Perf impact: 3.5%

Src: https://github.com/firezone/firezone/pull/6673

READ & WRITE IP PACKETS IN THREADS

```
let mut state = AppState::new();

// Spawn a TUN thread and return channels
let (mut tun_tx, mut tun_rx) = ThreadedTun::new(...);

// Spawn a UDP thread and return channels
let (mut udp_tx, mut udp_rx) = ThreadedUdpSocket::bind(...);

loop {
    let packet = select(
        udp_rx.next_packet(),
        tun_rx.next_packet()
    ).await;

    let packet = state.handle(packet);

    match packet {
        Packet::Udp(p) => udp_tx.send(p).await,
        Packet::Ip(p) => tun_tx.send(p).await
    }
}
```

CAN WE AVOID THE COPYING?

CAN WE AVOID THE COPYING?

- Buffer pools!
  - Most difficult part is fragmentation (i.e. bin-packing problem)
  - Same size -> No fragmentation

CAN WE AVOID THE COPYING?

- Buffer pools!
  - Most difficult part is fragmentation (i.e. bin-packing problem)
  - Same size -> No fragmentation
- Bounded channels -> bounded memory

MAKE SYSCALLS CHEAPER

- GRO (Generic Receive Offload)
- GSO (Generic Segmentation Offload)

MAKE SYSCALLS CHEAPER

- GRO (Generic Receive Offload)
- GSO (Generic Segmentation Offload)
- Don't make one syscall for each packet

MAKE SYSCALLS CHEAPER

- GRO (Generic Receive Offload)
- GSO (Generic Segmentation Offload)
- Don't make one syscall for each packet
- Batch read & write packets with the same headers and payload (segment) length

MAKE SYSCALLS CHEAPER

- GRO (Generic Receive Offload)
- GSO (Generic Segmentation Offload)
- Don't make one syscall for each packet
- Batch read & write packets with the same headers and payload (segment) length
- Only works for UDP

MAKE SYSCALLS CHEAPER

- GRO (Generic Receive Offload)
- GSO (Generic Segmentation Offload)
- Don't make one syscall for each packet
- Batch read & write packets with the same headers and payload (segment) length
- Only works for UDP

```
| Header | payload (1316) + payload (1316) + payload (588)
```

GRO (FOR UDP) IS EASY

- Pass a single buffer to socket
- Perform GRO syscall
- Segment resulting buffer accordingly
- Process packets in a loop

```rust
let mut state = AppState::new();
let (mut tun_tx, mut tun_rx) = ThreadedTun::new(...);
let (mut udp_tx, mut udp_rx) = ThreadedUdpSocket::bind(...);

loop {
    let packet = select(
        udp_rx.next_packet(),
        tun_rx.next_packet()
    ).await;

    match packet {
        Packet::Udp(p_iter) => {
            for p in p_iter {
                let ip_packet = state.handle_udp(p);
                tun_tx.send(ip_packet).await
            }
        },
        Packet::Ip(p) => {
            let udp_packet = state.handle_ip(p);
            udp_tx.send(udp_packet).await
        }
    }
}
```

WHAT ABOUT GSO?

WHAT ABOUT GSO?

- In order to send a batch of UDP packets, we need a batch of IP packets

WHAT ABOUT GSO?

- In order to send a batch of UDP packets, we need a batch of IP packets
- `Receiver::recv_many(&mut Vec<T>, usize)`

WHAT ABOUT GSO?

- In order to send a batch of UDP packets, we need a batch of IP packets
- `Receiver::recv_many(&mut Vec<T>, usize)`

```
let gso_queue = GsoQueue::new();
let ip_batch: Vec<IpPacket> = tun_rx.next_batch().await;

for p in ip_batch {
    let udp_packet = state.handle_ip(p);
    gso_queue.submit(udp_packet);
}

for batch in gso_queue.batches() {
    udp_tx.send(udp_packet).await
}
```

TOTAL PERF IMPROVEMENT

~380 MBit/s => ~2.1 GBit/s

FUTURE IDEAS

- Cross-platform, completion-based IO (io-uring etc)
- SO_REUSEPORT: multi-threaded UDP socket
- GRO/GSO for TUN device

CONCLUSIONS

CONCLUSIONS

- copying memory is surprisingly cheap

CONCLUSIONS

- copying memory is surprisingly cheap
- syscalls are surprisingly expensive

CONCLUSIONS

- copying memory is surprisingly cheap
- syscalls are surprisingly expensive
- references can be surprisingly limiting

CONCLUSIONS

- copying memory is surprisingly cheap
- syscalls are surprisingly expensive
- references can be surprisingly limiting
- buffer pools (for fixed sized structures) are surprisingly easy

CONCLUSIONS

- copying memory is surprisingly cheap
- syscalls are surprisingly expensive
- references can be surprisingly limiting
- buffer pools (for fixed sized structures) are surprisingly easy
- Allocate on the heap, use a buffer pool once it becomes to expensive

CONCLUSIONS

- copying memory is surprisingly cheap
- syscalls are surprisingly expensive
- references can be surprisingly limiting
- buffer pools (for fixed sized structures) are surprisingly easy
- Allocate on the heap, use a buffer pool once it becomes to expensive
- Watch out for memory leaks / unbounded growth