

# Safe and Secure Drivers in High-Level Languages

**Paul Emmerich, Simon Ellmann, Sebastian Voit,**  
Fabian Bonk, Alex Egger, Alexander Frank, Thomas Günzel,  
Stefan Huber, Maximilian Pudelko, Maximilian Stadlmeier

December 29, 2018

Chair of Network Architectures and Services  
Department of Informatics  
Technical University of Munich

# Safe and Secure Drivers in High-Level Languages

**Paul Emmerich<sup>1</sup>, Simon Ellmann<sup>2</sup>, Sebastian Voit<sup>3</sup>,  
Fabian Bonk<sup>4</sup>, Alex Egger<sup>5</sup>, Alexander Frank<sup>6</sup>, Thomas Günzel<sup>7</sup>,  
Stefan Huber<sup>8</sup>, Maximilian Pudelko<sup>9</sup>, Maximilian Stadlmeier<sup>10</sup>**

<sup>1</sup>C, thesis advisor   <sup>2</sup>Rust   <sup>3</sup>Go   <sup>4</sup>OCaml   <sup>5</sup>Haskell  
<sup>6</sup>Latency measurements   <sup>7</sup>Swift   <sup>8</sup>IOMMU   <sup>9</sup>VirtIO driver   <sup>10</sup>C#

Chair of Network Architectures and Services  
Department of Informatics  
Technical University of Munich

# About us

## Paul

- PhD student at Technical University of Munich
- Researching performance of packet processing systems



## Simon

- Rust driver as bachelor's thesis, now HiWi/research assistant



## Sebastian

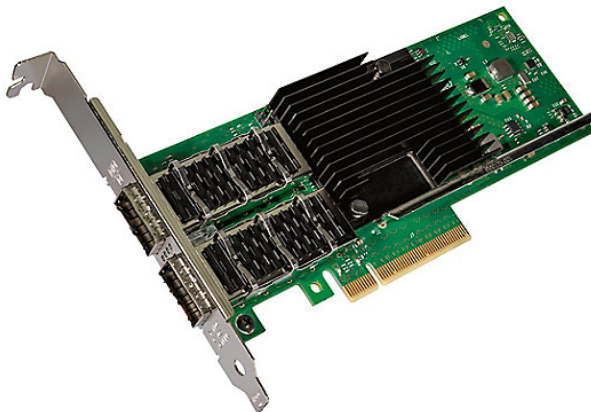
- Go driver as bachelor's thesis



## Everyone else mentioned on the title slide

- Did a thesis with Paul as advisor

# Network drivers



Intel XL710 [Picture: Intel.com]

# The ixy project

- Attempt to write a simple yet fast user space network driver
- It's a user space driver you can easily understand and read
- $\approx$  1,000 lines of C code, full of references to datasheets and specs
- Supports Intel ixgbe NICs
- New: supports VirtIO NICs (qemu/kvm and VirtualBox, we got a Vagrant setup!)
- Check it out on GitHub: <https://github.com/emmericp/ixy>

# Expectation: Beautiful C code

- Why write a driver in C?

# Expectation: Beautiful C code

- Why write a driver in C?
- Most drivers are written in C
- C is the lowest common denominator of systems programming languages
- C code can be beautiful
- Everyone can read C?

# Reality: C can be ugly

```
#define mystery_macro(ptr, type, member) ({\n    const typeof(((type*)0)->member)* __mptr = (ptr);\n    (type*)((char*)__mptr - offsetof(type, member));\n})
```



# Reality: C can be ugly

```
#define container_of(ptr, type, member) ({\n    const typeof(((type*)0)->member)* __mptr = (ptr);\n    (type*)((char *)__mptr - offsetof(type, member));\n})
```

## Reality: C can be ugly

```
#define container_of(ptr, type, member) ({\n    const typeof(((type*)0)->member)* __mptr = (ptr);\n    (type*)((char *)__mptr - offsetof(type, member));\n})
```

- Allows some “inheritance” in C to abstract driver implementations
- Virtually all C drivers use this macro
- The Linux kernel contains  $\approx 15,000$  uses of this macro

# C can cause security problems

Vulnerability Trends Over Time

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges
<a href="#">1999</a>	19	<a href="#">2</a>		<a href="#">3</a>						<a href="#">1</a>		<a href="#">2</a>
<a href="#">2000</a>	5	<a href="#">3</a>										<a href="#">1</a>
<a href="#">2001</a>	22	<a href="#">6</a>								<a href="#">4</a>		<a href="#">3</a>
<a href="#">2002</a>	15	<a href="#">3</a>		<a href="#">1</a>						<a href="#">1</a>	<a href="#">1</a>	
<a href="#">2003</a>	19	<a href="#">8</a>		<a href="#">2</a>						<a href="#">1</a>	<a href="#">3</a>	<a href="#">4</a>
<a href="#">2004</a>	51	<a href="#">20</a>	<a href="#">5</a>	<a href="#">12</a>							<a href="#">5</a>	<a href="#">12</a>

(...)

<a href="#">2017</a>	454	<a href="#">147</a>	<a href="#">169</a>	<a href="#">52</a>	<a href="#">26</a>			<a href="#">1</a>		<a href="#">17</a>	<a href="#">89</a>	<a href="#">36</a>
<a href="#">2018</a>	166	<a href="#">81</a>	<a href="#">3</a>	<a href="#">28</a>	<a href="#">8</a>					<a href="#">3</a>	<a href="#">17</a>	<a href="#">3</a>
Total	2155	<a href="#">1184</a>	<a href="#">241</a>	<a href="#">347</a>	<a href="#">124</a>			<a href="#">3</a>		<a href="#">111</a>	<a href="#">350</a>	<a href="#">260</a>
% Of All		54.9	11.2	16.1	5.8	0.0	0.0	0.1	0.0	5.2	16.2	12.1

- Screenshot from <https://www.cvedetails.com/>
- Security bugs found in the Linux kernel in the last  $\approx 20$  years

# C can cause security problems

- Not all bugs can be blamed on the language
- Cutler et al. analyzed 65 CVEs categorized as code execution in the Linux kernel <sup>1</sup>

---

<sup>1</sup> C. Cutler, M. F. Kaashoek, and R. T. Morris, “[The benefits and costs of writing a POSIX kernel in a high-level language](#)”, USENIX OSDI, 2018

## C can cause security problems

- Not all bugs can be blamed on the language
- Cutler et al. analyzed 65 CVEs categorized as code execution in the Linux kernel <sup>1</sup>

Bug type	Num.	Perc.	Can be avoided by language?
Various	11	17%	Unclear/Maybe
Logic	14	22%	No
Use-after-free	8	12%	Yes
Out of bounds	32	49%	Yes (likely leads to panic)

**Table 1:** Code execution vulnerabilities in the Linux kernel identified by Cutler et al<sup>1</sup>

<sup>1</sup> C. Cutler, M. F. Kaashoek, and R. T. Morris, “[The benefits and costs of writing a POSIX kernel in a high-level language](#)”, USENIX OSDI, 2018

# Are there preventable bugs in drivers?

- We looked at these 40 preventable bugs

# Are there preventable bugs in drivers?

- We looked at these 40 preventable bugs
- 39 of them were in drivers (the other was in the Bluetooth stack)

# Are there preventable bugs in drivers?

- We looked at these 40 preventable bugs
- 39 of them were in drivers (the other was in the Bluetooth stack)
- 13 were in the Qualcomm WiFi driver



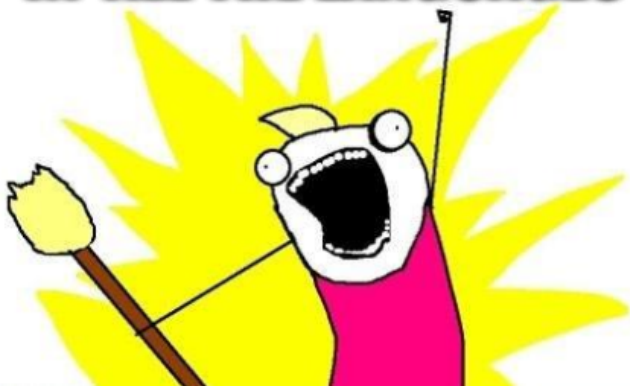
# Should you really write new code in C in 2019?

- If you have a choice: probably not, no

# Should you really write new code in C in 2019?

- If you have a choice: probably not, no
- User space drivers can be written in **any** language!
- But are all languages an equally good choice?
- Is a JIT compiler or a garbage collector a problem in a driver?











**WRITE DRIVERS  
IN ALL THE LANGUAGES**



**ALL THE LANGUAGES?**



## OPEN

Title	Type	Advisors	Year	Links
Writing Network Drivers in Rust	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Go	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Java	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in C#	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Haskell	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Scala	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in OCaml	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Javascript	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Python	<a href="#">BA</a> , <a href="#">MA</a> , <a href="#">IDP</a>	<a href="#">Paul Emmerich</a>	2018	
Writing Network Drivers in Bash	<a href="#">BA</a>	<a href="#">Paul Emmerich</a>	2018	

# Basics: How to talk to (modern) PCIe devices

1. Memory-mapped IO (MMIO)
2. Direct memory access (DMA)
3. Interrupts

# Basics: How to talk to (modern) PCIe devices

## 1. Memory-mapped IO (MMIO)

- Magic memory area that is mapped to the device
- Memory reads/writes are directly forwarded to the device
- Usually used to expose device registers
- User space drivers: `mmap` a magic file

## 2. Direct memory access (DMA)

## 3. Interrupts

# Basics: How to talk to (modern) PCIe devices

1. Memory-mapped IO (MMIO)
2. Direct memory access (DMA)
  - Allows the device to read/write **arbitrary** memory locations
  - User space drivers: figure out physical addresses, tell the device to write there
3. Interrupts



# Basics: How to talk to (modern) PCIe devices

1. Memory-mapped IO (MMIO)

2. Direct memory access (DMA)

3. Interrupts

- This is how the device informs you about events
- User space drivers: available via the Linux `vfio` subsystem
- (Usually) not useful for high-speed network drivers
- We'll ignore interrupts here

# Basics: How to write a user space driver in 4 simple steps

1. Unload kernel driver
2. mmap the PCIe MMIO address space
3. Figure out physical addresses for DMA
4. Write the driver

## Hardware: Intel ixgbe family (10 Gbit/s)

- ixgbe family: 82599ES (aka X520), X540, X550, Xeon D embedded NIC
- Commonly found in servers or as on-board chips
- Very good datasheet publicly available
- Almost no logic hidden behind black-box firmware

## Hardware: Intel ixgbe family (10 Gbit/s)

- ixgbe family: 82599ES (aka X520), X540, X550, Xeon D embedded NIC
- Commonly found in servers or as on-board chips
- Very good datasheet publicly available
- Almost no logic hidden behind black-box firmware
- Drivers for many newer NICs often just exchanges messages with the firmware
- Here: all hardware features directly exposed to the driver

# Find the device we want to use

```
# lspci
03:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
03:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
```

# Find the device we want to use

```
# lspci
03:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
03:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ ...
```

# Unload the kernel driver

```
echo 0000:03:00.1 > /sys/bus/pci/devices/0000:03:00.1/driver/unbind
```

## mmap the PCIe register address space from user space

```
int fd = open("/sys/bus/pci/devices/0000:03:00.0/resource0", O_RDWR);
struct stat stat;
fstat(fd, &stat);
uint8_t* registers = (uint8_t*) mmap(NULL, stat.st_size, PROT_READ | PROT_WRITE,
                                     MAP_SHARED, fd, 0);
```



# Device registers

**Table 8-2 Register Summary**

Offset / Alias Offset	Abbreviation	Name	Block	RW	Reset Source	Page
<b>General Control Registers</b>						
0x00000 / 0x00004	CTRL	Device Control Register	Target	RW		543
0x00008	STATUS	Device Status Register	Target	RO		544
0x00018	CTRL_EXT	Extended Device Control Register	Target	RW		544
0x00020	ESDP	Extended SDP Control	Target	RW		545
0x00028	I2CCTL	I2C Control	Target	RW	PERST	549
0x00200	LEDCTL	LED Control	Target	RW		549
0x05078	EXVET	Extended VLAN Ether Type	Target	RW		551

# Access registers: LEDs

```
#define LEDCTL 0x00200
```

```
#define LED0_BLINK_OFFS 7
```

```
uint32_t leds = *((volatile uint32_t*)(registers + LEDCTL));
```

```
*((volatile uint32_t*)(registers + LEDCTL)) = leds | (1 << LED0_BLINK_OFFS);
```

- Memory-mapped IO: all memory accesses go directly to the NIC
- One of the very few valid uses of **volatile** in C

# Handling packets via DMA

- Packets are transferred via queue interfaces (often called rings)
- Rings are configured via MMIO and accessed by the device via DMA
- Rings (usually) contain pointers to packets, also accessed via DMA

# Handling packets via DMA

- Packets are transferred via queue interfaces (often called rings)
- Rings are configured via MMIO and accessed by the device via DMA
- Rings (usually) contain pointers to packets, also accessed via DMA
- Details vary between different devices
- This is not unique to NICs: most PCIe devices work in a similar manner

# Challenges for high-level languages

- Access to `mmap` with the proper flags
- Handle externally allocated (foreign) memory in the language
- Handle memory layouts/formats (i.e., access memory that looks like a given C struct)
- Memory access semantics: memory barriers, volatile reads/writes
- Some operations in drivers are inherently unsafe

# Goals for our implementations

- Implement the same feature set as my C reference driver
- Use a similar structure like the C driver
- Write idiomatic code for the selected language
- Use language safety features where possible
- Quantify trade-offs for performance vs. safety

# C#

# C#

- No, we didn't develop a Windows driver
- We used Microsoft's CoreCLR on Linux



# C#

- No, we didn't develop a Windows driver
- We used Microsoft's CoreCLR on Linux
- JIT compiled
- Garbage collected
- Memory safe (mostly)

# C#

- No, we didn't develop a Windows driver
- We used Microsoft's CoreCLR on Linux
- JIT compiled
- Garbage collected
- Memory safe (mostly)
- C# supports a relatively obscure [unsafe mode](#)
- Unsafe mode features full support for pointers

## C#: Access to external memory

- C# provides `UnmanagedMemoryStream`, a nice wrapper for foreign memory
- But it was too slow :(

## C#: Access to external memory

- C# provides `UnmanagedMemoryStream`, a nice wrapper for foreign memory
- But it was too slow :(
- Use unsafe raw pointers for packet buffers instead

```
public unsafe void WriteData(uint offset, int val) {  
    if (offset >= BUF_SIZE) throw new IndexOutOfRangeException();  
    volatile int *ptr = (volatile int*)(_baseAddress + DataOffset + offset);  
    *ptr = val;  
}
```

## C#: Access to external memory

- C# provides `UnmanagedMemoryStream`, a nice wrapper for foreign memory
- But it was too slow :(
- Use unsafe raw pointers for packet buffers instead

```
public unsafe void WriteData(uint offset, int val) {  
    if (offset >= BUF_SIZE) throw new IndexOutOfRangeException();  
    volatile int *ptr = (volatile int*)(_baseAddress + DataOffset + offset);  
    *ptr = val;  
}
```

- Looks a lot like C
- Potentially unsafe operations are all in a few known places, simpler auditing



Swift

# Swift

- No, we didn't develop a macOS/iOS driver
- Swift is also available on Linux

# Swift

- No, we didn't develop a macOS/iOS driver
- Swift is also available on Linux
- Compiled via LLVM
- Memory management via reference counting (ARC)
- Memory safe (mostly)



## Swift: Pointers

- UnsafeBufferPointer and co wrap foreign memory blobs
- Used to make packets in DMA buffers available

```
public var packetData: UnsafeBufferPointer<UInt8>? {  
    get {  
        return UnsafeBufferPointer<UInt8>(  
            start: self.entry.pointer.assumingMemoryBound(to: UInt8.self),  
            count: Int(self.size)  
        )  
    }  
}
```

- Forces you to specify the buffer size, accesses check the bounds in debug mode

## Swift: Pointers with a very verbose syntax

- Example: modify one byte in a packet (part of our benchmark)

```
public func touch() {  
    let ptr = self.entry.pointer  
    var newValue: UInt32 = ptr.load(fromByteOffset: 0, as: UInt32.self)  
    newValue += 1  
    ptr.storeBytes(of: newValue, toByteOffset: 0, as: UInt32.self)  
}
```

- Quite verbose compared to C or C#



**OCaml**

# OCaml

- Compiled language
- Memory management via garbage collection
- Memory safe
- Functional language

# OCaml: Cstruct

```
[%%cstruct
```

```
  type adv_rxd_wb = {  
    pkt_info : uint16;  
    hdr_info : uint16;  
    ip_id : uint16;  
    csum : uint16;  
    status_error : uint32;  
    length : uint16;  
    vlan : uint16  
  } [@@little_endian]
```

```
]
```

- Cstruct generates accessors to work with (foreign) memory that looks like this

## OCaml: It looks quite different

- Code that checks how many packets are ready to be read in the receive ring

```
let num_done =  
  (* counting without mutation *)  
  let rec loop offset =  
    let rxd = descriptors.(wrap_rx (rxq.rx_index + offset)) in  
    if Int32.((get_adv_rx_wb_status rxd) land RXD.stat_dd <> 01) then  
      loop (offset + 1)  
    else  
      offset in  
  loop 0
```



# Haskell

- Compiled language (GHC)
- Memory management via garbage collection
- Memory safe
- Functional language



# Haskell: Access to foreign memory

- `mmap` and `mlock` available via `System.Posix.Memory`
  - All necessary flags and features are available in Haskell, we had to write some C code to get `mmap/mlock` in OCaml
- Foreign package provides access to foreign memory

# Haskell: Sum types are useful

- Descriptors often exist in two forms
  - One format written by the driver and read by the device
  - A second format that is written back by the device once it's finished

```
data TransmitDescriptor = TransmitRead { tdBufPhysAddr :: !Word64
                                         , tdCmdTypeLen  :: !Word32
                                         , tdOlInfoStatus :: !Word32 }
  | TransmitWriteback { tdStatus :: !Word32 }
```



GO

The image features the word "GO" in a bold, teal-colored, sans-serif typeface. To the left of the "G", there are three horizontal teal lines of varying lengths, stacked vertically, which serve as motion lines to suggest speed or forward movement. The entire logo is set against a plain white background.

# Go

- Compiled programming language developed by Google
- General purpose language but designed for distributed systems

# Go

- Compiled programming language developed by Google
- General purpose language but designed for distributed systems
- A driver is not a distributed system

# Go

- Compiled programming language developed by Google
- General purpose language but designed for distributed systems
- A driver is not a distributed system
- Then why even use Go?

# Go

- Compiled programming language developed by Google
- General purpose language but designed for distributed systems
- A driver is not a distributed system
- Then why even use Go?
  - Runtime for:  
Garbage Collection  
Memory & Type safety
  - Large standard library

# Go for drivers

- Actually a lot like C in many aspects



# Go for drivers

- Actually a lot like C in many aspects
- Main differences:
  - No pointer arithmetic (managing DMA memory)
  - No volatile (memory barriers for register access)

# Go for drivers

- Actually a lot like C in many aspects
- Main differences:
  - No pointer arithmetic (managing DMA memory)
  - No volatile (memory barriers for register access)
- What we do instead:
  - Manage DMA memory via slices
  - Unsafe pointers: circumvent runtime but allow arbitrary pointer  
→ Physical address calculation & register access
  - Rule set for unsafe pointers to still be valid

# Managing memory: mempools

- `syscall.Mmap()` returns slice of the mmaped memory area
- For this presentation: slice = fancy array  
→ bounds checked, subslicing, etc.

```
//allocate DMA memory & initialize mempool
for i := uint32(0); i < numEntries; i++ {
    mempool.packetBuf[i] = &PktBuf{
        Pkt:      mempool.buf[i*entrySize : (i+1)*entrySize],
        PhyAddr:   uint64(virtToPhys(uintptr(
            unsafe.Pointer(&mempool.buf[i*entrySize])))),
    }
}
```

## No volatile, no problem

- Registers share memory with NIC
- Compiler memory barrier to prevent re-ordering
- sync/atomic functions prevent re-ordering around them

```
func setReg32(addr []byte, reg int, value uint32) {  
    atomic.StoreUint32((*uint32)(unsafe.Pointer(&addr[reg])), value)  
}
```

```
func getReg32(addr []byte, reg int) uint32 {  
    return atomic.LoadUint32((*uint32)(unsafe.Pointer(&addr[reg])))  
}
```

# Conclusion Go

- Actually quite nice to work with
  - Safety (see Cutler et al.<sup>2</sup>)
  - Looks like C in beautiful

---

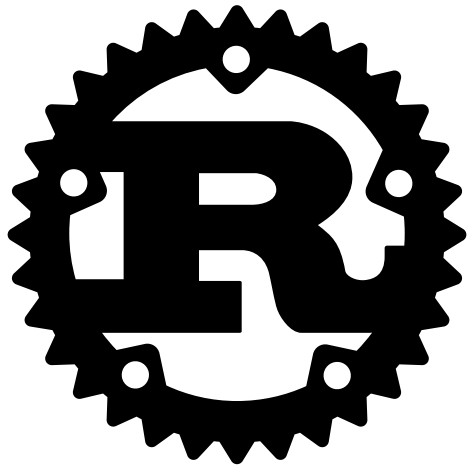
<sup>2</sup> C. Cutler, M. F. Kaashoek, and R. T. Morris, “[The benefits and costs of writing a POSIX kernel in a high-level language](#)”, USENIX OSDI, 2018

# Conclusion Go

- Actually quite nice to work with
  - Safety (see Cutler et al.<sup>2</sup>)
  - Looks like C in beautiful
- But
  - Approx 10% slower then C
  - Descriptor access can be ugly (functions on descriptors are too costly)

---

<sup>2</sup> C. Cutler, M. F. Kaashoek, and R. T. Morris, “[The benefits and costs of writing a POSIX kernel in a high-level language](#)”, USENIX OSDI, 2018



# Rust

## What is Rust?

*A safe, concurrent, practical systems language.*

Great! That's what we are looking for! Anything else we need to know?

- No garbage collector
- Unique ownership system and rules for moving/borrowing values
- Unsafe mode



# Safety in Rust: The ownership system

- Three rules:
  1. Each value has a variable that is its owner
  2. There can only be one owner at a time
  3. When the owner goes out of scope, the value is freed
- Rules enforced at compile-time
- Ownership can be passed to another variable

## Safety in Rust: The ownership system by example

- Packets are owners of some DMA memory
- Packets are passed between users and the driver, thus ownership is passed as well
- At any point in time there is only one Packet owner that can change its memory

```
let buffer: &mut VecDeque<Packet> = VecDeque::new();
dev.rx_batch(RX_QUEUE, buffer, BATCH_SIZE);
for p in buffer.iter_mut() {
    p[48] += 1;
}
dev.tx_batch(TX_QUEUE, buffer);
buffer.drain(..);
```

# Safety in Rust: Unsafe code

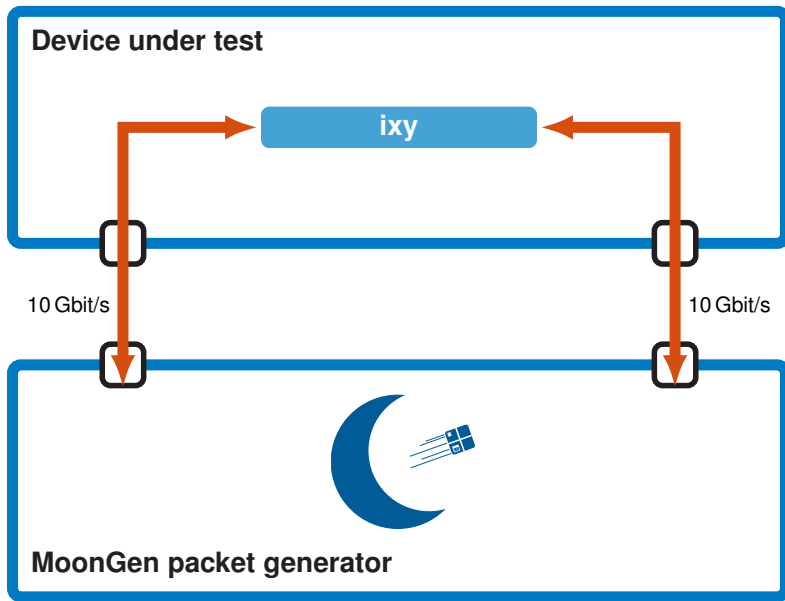
- Not everything can be done in safe Rust
- Calling foreign functions and dereferencing raw pointers is per se unsafe
- Many functions in Rust's standard library make use of unsafe code

```
let ptr = unsafe {  
    libc::mmap(  
        ptr::null_mut(), len, libc::PROT_READ | libc::PROT_WRITE,  
        libc::MAP_SHARED, file.as_raw_fd(), 0,  
    ) as *mut u8  
};
```

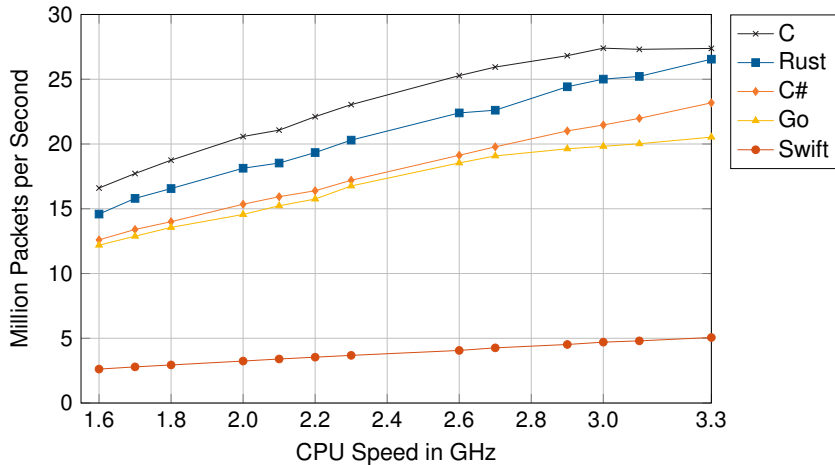
# Rust by example

- Biggest challenge: safe memory handling with unsafe code

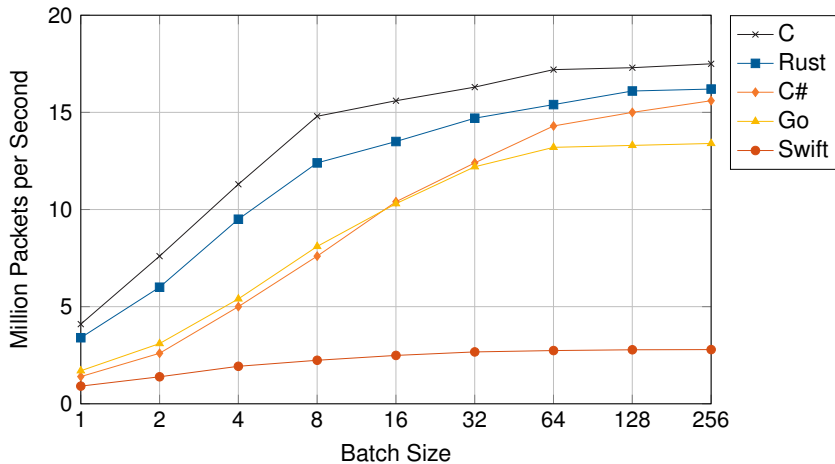
```
fn set_reg32(&self, reg: u32, val: u32) {  
    assert!(  
        reg as usize <= self.len - 4 as usize,  
        "memory access out of bounds"  
    );  
  
    unsafe {  
        ptr::write_volatile((self.addr as usize + reg as usize) as *mut u32, val);  
    }  
}
```



## Performance comparison (single CPU core)



## Batching at 1.6 GHz CPU speed



## Swift: Flame graph



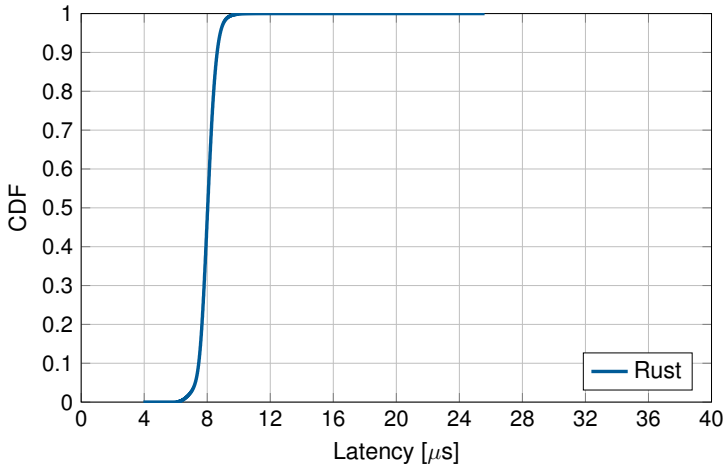
# Swift: Why so slow?

- Lots of time spent in Swift's memory management
- Swift adds calls to release/retain for each used object in each function
- This is basically the same as wrapping every object in a `std::shared_ptr` in C++

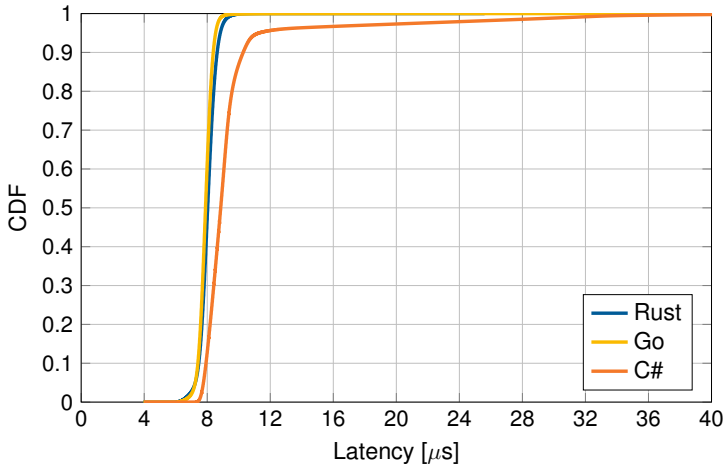
# Swift: Why so slow?

- Lots of time spent in Swift's memory management
- Swift adds calls to release/retain for each used object in each function
- This is basically the same as wrapping every object in a `std::shared_ptr` in C++
- Time in release/retain: 76%
- For comparison: Go spends less than 0.5% in the garbage collector

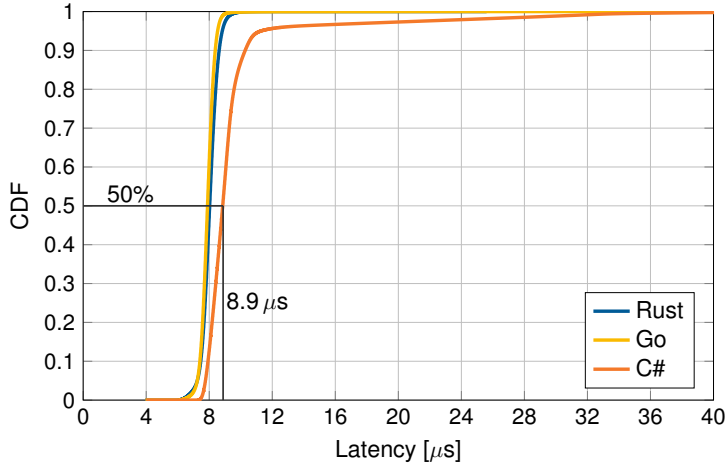
## Garbage collection and JIT compilation vs. latency



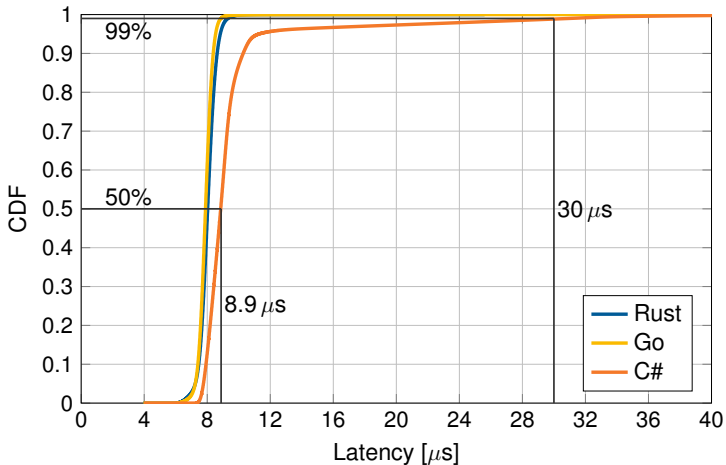
## Garbage collection and JIT compilation vs. latency



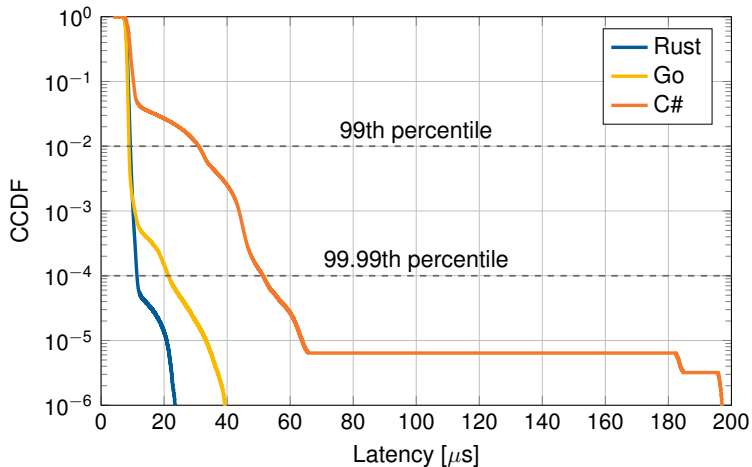
# Garbage collection and JIT compilation vs. latency



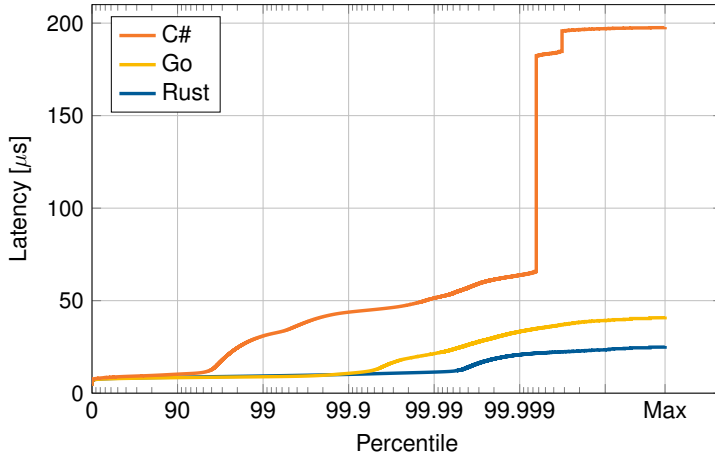
## Garbage collection and JIT compilation vs. latency



# Complementary cumulative distribution function



# Tail latency





# Look ma, no root

- User space drivers usually run with root privileges, but why?

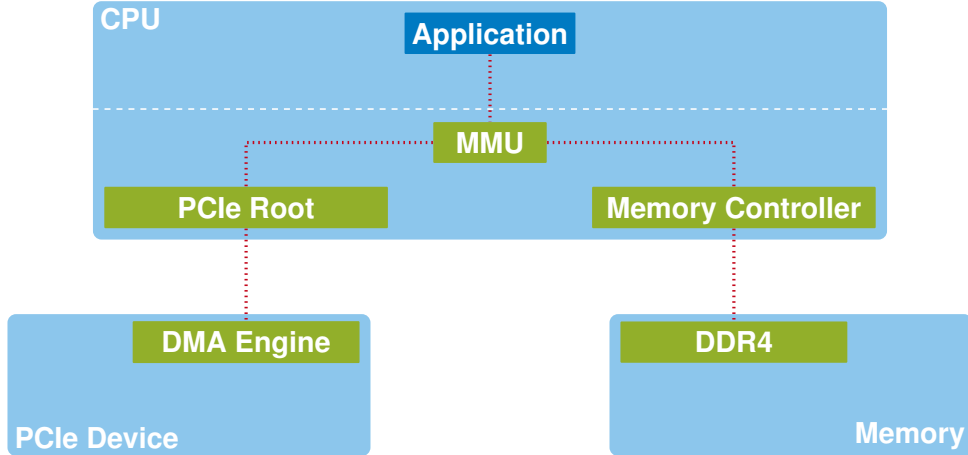
# Look ma, no root

- User space drivers usually run with root privileges, but why?
- Mapping PCIe resources requires root
- Allocating non-transparent huge pages requires root
- Locking memory requires root
- Can we do that in a small separate program that is easy to audit and then drop privileges?

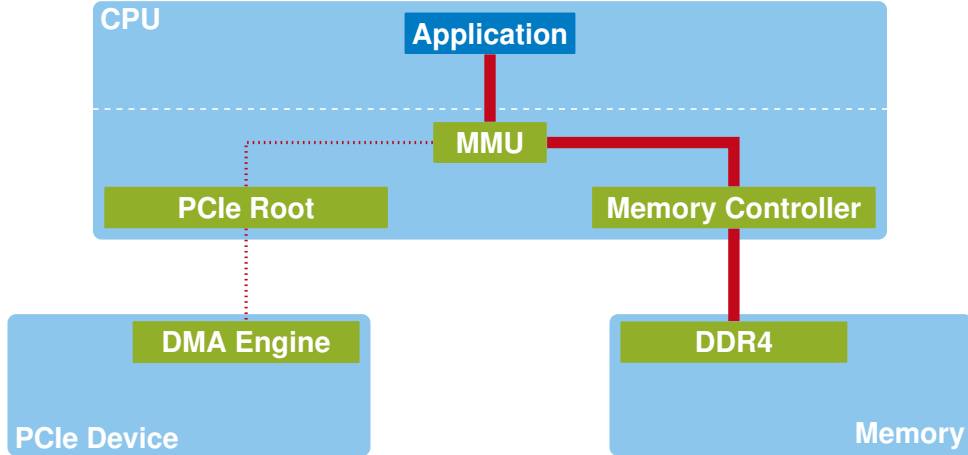
# Look ma, no root

- User space drivers usually run with root privileges, but why?
- Mapping PCIe resources requires root
- Allocating non-transparent huge pages requires root
- Locking memory requires root
- Can we do that in a small separate program that is easy to audit and then drop privileges?
- Yes, we can
- But it's not really secure

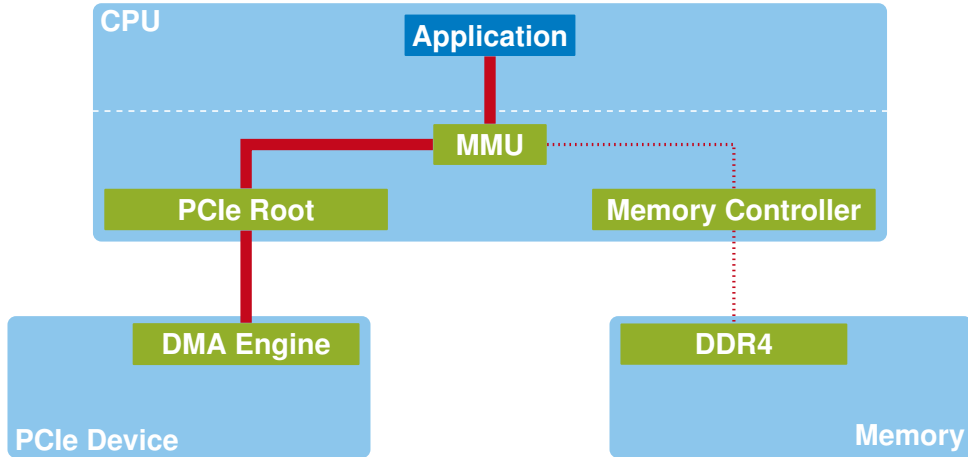
# Memory access on modern systems



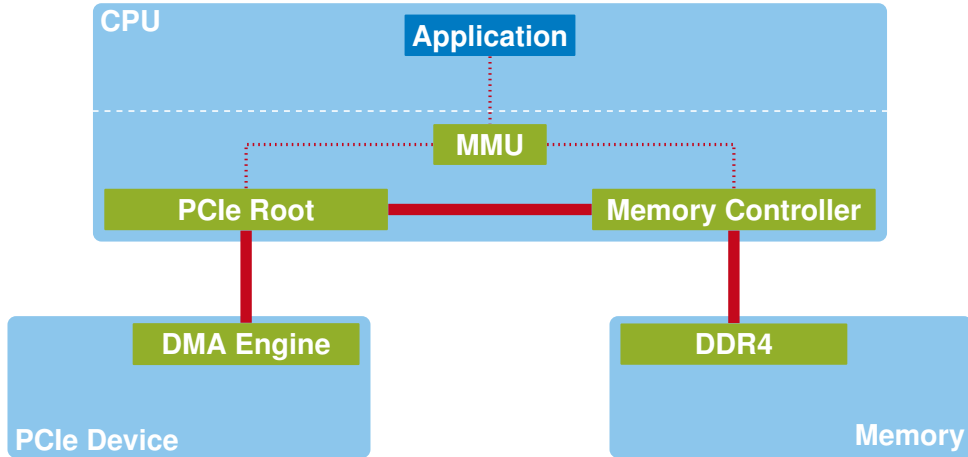
# Memory access on modern systems



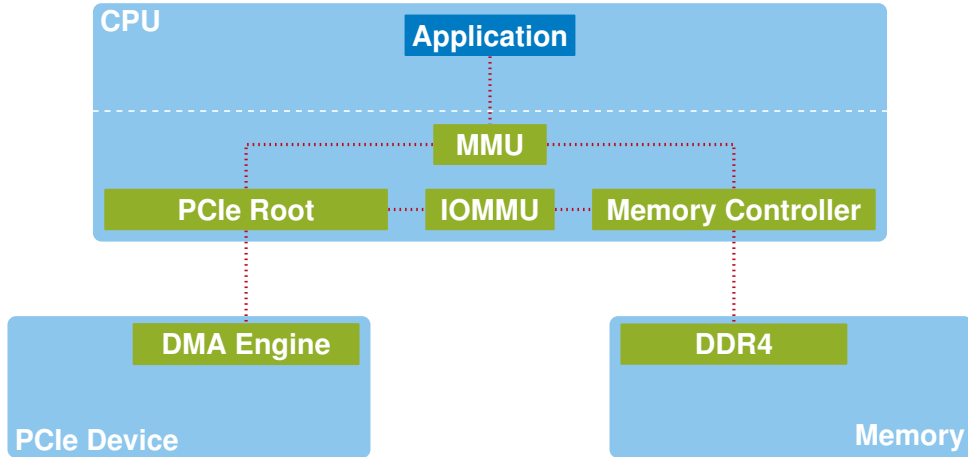
# Memory access on modern systems



# Memory access on modern systems

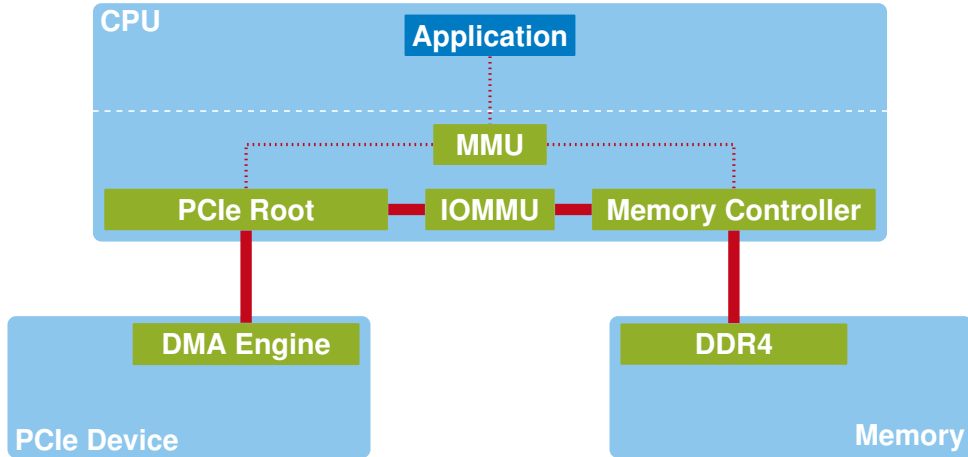


# Memory access on modern systems





# Memory access on modern systems



# Unprivileged user space drivers on Linux

1. Prepare the system as root
  - 1.1. Bind the device to the special `vfio` driver
  - 1.2. `chown` the special magic `vfio` device to your user
  - 1.3. Allow your user to lock some amount of memory via `ulimit`

# Unprivileged user space drivers on Linux

1. Prepare the system as root
  - 1.1. Bind the device to the special `vfio` driver
  - 1.2. `chown` the special magic `vfio` device to your user
  - 1.3. Allow your user to lock some amount of memory via `ulimit`
2. `mmap` the special magic `vfio` device
3. Do some magic `ioctl` calls on the magic device
4. Protected DMA memory can also be allocated via an `ioctl` call
5. Use the device as usual, all accesses are now checked by the IOMMU

# Unprivileged user space drivers on Linux

1. Prepare the system as root
    - 1.1. Bind the device to the special `vfio` driver
    - 1.2. `chown` the special magic `vfio` device to your user
    - 1.3. Allow your user to lock some amount of memory via `ulimit`
  2. `mmap` the special magic `vfio` device
  3. Do some magic `ioctl` calls on the magic device
  4. Protected DMA memory can also be allocated via an `ioctl` call
  5. Use the device as usual, all accesses are now checked by the IOMMU
- We have implemented this in our C driver, Rust is WIP

# Why write a user space network driver?

- Why not? It can be fun
- Maybe you need a quick & dirty driver for a weird device?
- Maybe you need quick development cycles while playing around with a custom device
- Maybe you need some feature not supported by the original driver

## Example: Hardware timestamping

- Our latency measurement requires timestamps with nanosecond-level precision
- It also needs to handle millions of packets per second (we measured with  $\approx 15$  Mpps)
- This usually requires special hardware (we've used NetFPGAs to do this in the past)

## Example: Hardware timestamping

- Our latency measurement requires timestamps with nanosecond-level precision
- It also needs to handle millions of packets per second (we measured with  $\approx 15$  Mpps)
- This usually requires special hardware (we've used NetFPGAs to do this in the past)
- Some cheap off-the-shelf NICs can add a timestamp to all incoming packets
- But none of the existing drivers support this feature :(

## Example: Hardware timestamping

- Our latency measurement requires timestamps with nanosecond-level precision
- It also needs to handle millions of packets per second (we measured with  $\approx 15$  Mpps)
- This usually requires special hardware (we've used NetFPGAs to do this in the past)
- Some cheap off-the-shelf NICs can add a timestamp to all incoming packets
- But none of the existing drivers support this feature :(
- We just set some flags in the right registers and got precise timestamping for cheap
- We used a Xeon D embedded NIC capturing all packets via a fiber optic splitter before and after our device under test (precision  $\approx \pm 15$  ns)



## Conclusion: Check out our code



- Meta-repository with links: <https://github.com/ixy-languages/ixy-languages>
- Drivers are simple: don't be afraid of them
- No kernel code needed :)