

# Drivers in High-Level Languages

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

21. April 2019

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

# Drivers in High-Level Languages

**Paul Emmerich<sup>1</sup>, Simon Ellmann<sup>2</sup>, Fabian Bonk<sup>3</sup>, Alex Egger<sup>4</sup>,  
Alexander Frank<sup>5</sup>, Thomas Günzel<sup>6</sup>, Stefan Huber<sup>7</sup>, Alexandru Obada<sup>8</sup>,  
Maximilian Pudelko<sup>9</sup>, Maximilian Stadlmeier<sup>10</sup>, Sebastian Voit<sup>11</sup>**

<sup>1</sup>C, Thesis advisor   <sup>2</sup>Rust   <sup>3</sup>OCaml   <sup>4</sup>Haskell   <sup>5</sup>Latency measurement setup  
<sup>6</sup>Swift   <sup>7</sup>IOMMU   <sup>8</sup>Python   <sup>9</sup>VirtIO Treiber   <sup>10</sup>C#   <sup>11</sup>Go

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

# About us

## Paul

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



## Simon

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



## Everyone else mentioned on the title slide

- Did a thesis with Paul as advisor

# C is an awesome language for operating systems!

- Low-level access to memory and devices
- Pointers are awesome
- You can write safe and secure code if you try really hard
- Everyone can read and write C
- C code can be beautiful

# Beautiful C code

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

# Beautiful C code

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

# Beautiful C code

```
#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 using a better 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

# Let's rewrite all operating systems in better languages?

- Rewriting the whole operating system in a safer language is a laudable effort
  - Redox (Rust) wants to become a production-grade OS but currently isn't
  - Singularity (Sing#, Microsoft Research) demonstrated some interesting concepts
  - Biscuit (Go) implements parts of POSIX for research
  - Unikernels like MirageOS (OCaml) or IncludeOS (C++) can be useful in some scenarios

# Let's rewrite all operating systems in better languages?

- Rewriting the whole operating system in a safer language is a laudable effort
  - Redox (Rust) wants to become a production-grade OS but currently isn't
  - Singularity (Sing#, Microsoft Research) demonstrated some interesting concepts
  - Biscuit (Go) implements parts of POSIX for research
  - Unikernels like MirageOS (OCaml) or IncludeOS (C++) can be useful in some scenarios
- But none of these will replace your main operating system any time soon

# Where are these bugs that could have been prevented?

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

# Where are these bugs that could have been prevented?

- 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

# Where are these bugs that could have been prevented?

- 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



# Can we rewrite drivers in better languages?

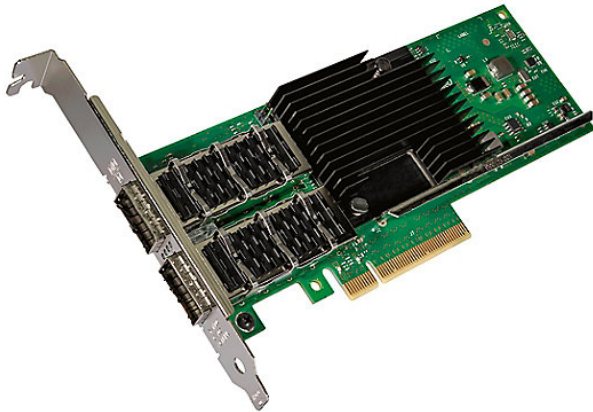
- Some operating systems have drivers in (subsets of) C++
- But good luck getting a driver in Rust or Go upstreamed in Linux



# Can we rewrite drivers in better languages?

- Some operating systems have drivers in (subsets of) C++
- But good luck getting a driver in Rust or Go upstreamed in Linux
- 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?

# Network drivers



Intel XL710 [Picture: Intel.com]

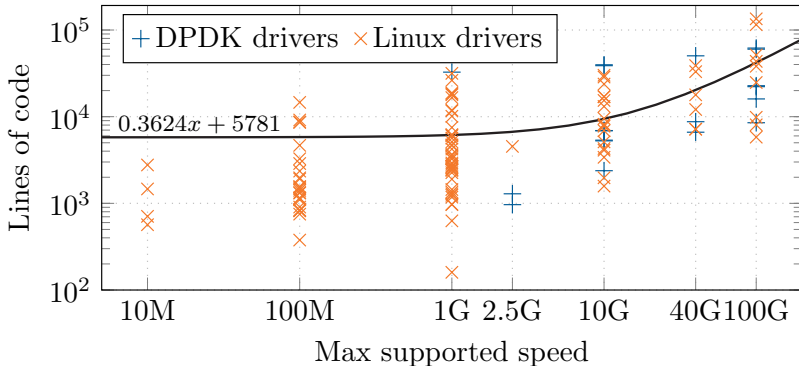
# Why look at network drivers?

- We happen to know a lot about networks ;)
- Easy to benchmark to quantify results
- Huge attack surface: exposed to the external world by design
- User space network drivers are already quite common (e.g., DPDK, Snabb)
- Network stacks are also moving into the user space (e.g., TCP stack on iOS)

# Why look at network drivers?

- We happen to know a lot about networks ;)
- Easy to benchmark to quantify results
- Huge attack surface: exposed to the external world by design
- User space network drivers are already quite common (e.g., DPDK, Snabb)
- Network stacks are also moving into the user space (e.g., TCP stack on iOS)
- Everything mentioned here is applicable to other drivers as well

## Network driver complexity is increasing



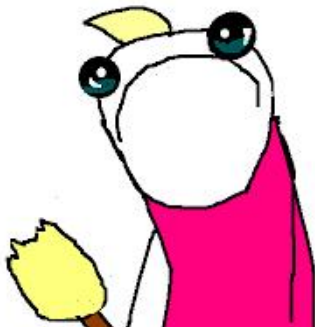
# The ixy driver

- Our attempt to write a simple yet fast user space network driver
- It's a user space driver you can easily understand and read
- Supports Intel ixgbe NICs (82599, X540, Xeon D, ...) and VirtIO
- $\approx$  1,000 lines of C code, full of references to datasheets and specs
- Intel driver: 38,000 lines in DPDK, 30,000 in Linux
- See talk “Demystifying Network Cards” at 34C3 for details
- But it's written in C, so let's rewrite that in a better and safer language

**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 (implementation is WIP)

# 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
- Black-box firmware contains almost no magic
- 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		<a href="#">543</a>
0x00008	STATUS	Device Status Register	Target	RO		<a href="#">544</a>
0x00018	CTRL_EXT	Extended Device Control Register	Target	RW		<a href="#">544</a>
0x00020	ESDP	Extended SDP Control	Target	RW		<a href="#">545</a>
0x00028	I2CCTL	I2C Control	Target	RW	PERST	<a href="#">549</a>
0x00200	LEDCTL	LED Control	Target	RW		<a href="#">549</a>
0x05078	EXVET	Extended VLAN Ether Type	Target	RW		<a href="#">551</a>

# 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

We wrote full user space drivers in these languages

C#



Swift



OCaml



# Goals for our implementations

- Implement the same feature set as our 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

# Language comparison: Overview

Language	Main paradigm	Memory management	Compilation
C	Imperative	No	Compiled
Rust	Imperative	Ownership/RAII	(LLVM) Compiled
Go	Imperative	Garbage collection	Compiled
C#	Object-oriented	Garbage collection	JIT
Swift	Protocol-oriented	Reference counting	(LLVM) Compiled
OCaml	Functional	Garbage collection	Compiled
Haskell	Functional	Garbage collection	(LLVM) Compiled
Python	Imperative	Garbage collection	Interpreted

Table 2: Language overview

## Language comparison: Safety properties

Language	General memory		Packet buffers		Int overflows
	Bounds checks	Use after free	Bounds checks	Use after free	
C	X	X	X	X	X
Rust					
Go					
C#					
Swift					
Haskell					
OCaml					
Python					

Table 3: Language-level protections against classes of bugs in our drivers

# Language comparison: Safety properties

Language	General memory		Packet buffers		Int overflows
	Bounds checks	Use after free	Bounds checks	Use after free	
C	✗	✗	✗	✗	✗
Rust	✓	✓	(✓) <sup>1</sup>	✓	(✓) <sup>4</sup>
Go	✓	✓	(✓) <sup>1</sup>	(✓) <sup>3</sup>	✗
C#	✓	✓	(✓) <sup>1</sup>	(✓) <sup>3</sup>	✗
Swift	✓	✓	✗ <sup>2</sup>	(✓) <sup>3</sup>	✓
Haskell	✓	✓	(✓) <sup>1</sup>	(✓) <sup>3</sup>	✗
OCaml	✓	✓	(✓) <sup>1</sup>	(✓) <sup>3</sup>	✗
Python	✓	✓	(✓) <sup>1</sup>	(✓) <sup>3</sup>	✗

<sup>1</sup> Bounds enforced by wrapper, constructor in unsafe code

<sup>2</sup> Bounds only enforced in debug mode

<sup>3</sup> Buffers are never free'd, only returned to a memory pool

<sup>4</sup> Disabled by default, proposed to be enabled by default in the future

Table 4: Language-level protections against classes of bugs in our drivers

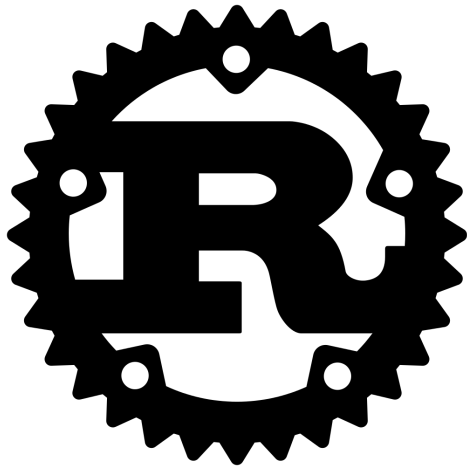
## Language comparison: Implementation sizes

Lang.	Lines of code <sup>1</sup>	Lines of C code <sup>1</sup>	Source size (gzip <sup>2</sup> )
C	831	831	12.9 kB
Rust	961	0	10.4 kB
Go	1640	0	20.6 kB
C#	1266	34	13.1 kB
Swift	1506	0	15.9 kB
Haskell	1001	0	9.6 kB
OCaml	1177	28	12.3 kB
Python	1242	(Cython) 77	14.2 kB

<sup>1</sup> Excluding empty lines and comments, counted with `cloc`

<sup>2</sup> Compression level 6

**Table 5:** Size of our implementations (w/o register constants, stripped features not found in all drivers)





# Rust

## What is Rust?

*A safe, concurrent, practical systems language.*

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

# Safety in Rust: The ownership system

- Immutability of variables by default
- 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
  - “moving” the value or by
  - “borrowing” it through a reference

## Safety in Rust: The ownership system by example

- Packets are owners of some DMA memory
- Packets are passed between user code 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 unsafe
- Many functions in Rust's standard library make use of unsafe code

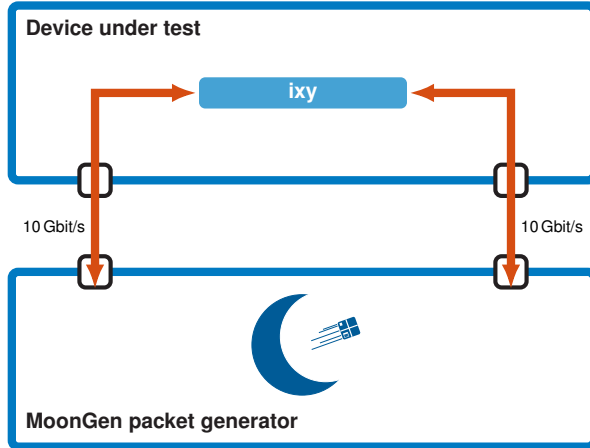
```
let ptr = unsafe {  
    libc::mmap(  
        ptr::null_mut(), len, libc::PROT_READ | libc::PROT_WRITE,  
        libc::MAP_SHARED, file.as_raw_fd(), 0,  
    ) as *mut u8  
};
```

## Example: Setting registers

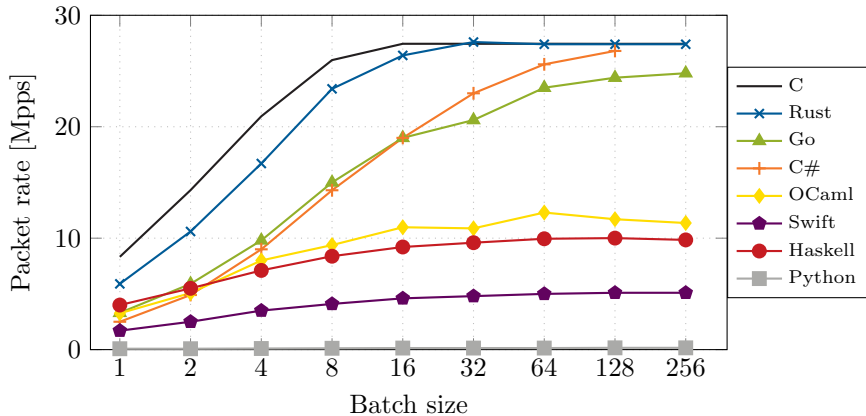
- Biggest challenge: safe memory handling with unsafe code

```
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: Test setup



## Batching at 3.3 GHz CPU speed (single core)



## 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

# Why is Rust slower than C?

Events per packet	Batch 32, 1.6 GHz		Batch 8, 1.6 GHz	
	C	Rust	C	Rust
<b>Cycles</b>	94	100	108	120
<b>Instructions</b>	127	209	139	232
<b>Instr. per cycle</b>	1.35	2.09	1.29	1.93
<b>Branches</b>	18	24	19	27
<b>Branch mispredicts</b>	0.05	0.08	0.02	0.06
<b>Store <math>\mu</math>ops</b>	21.8	37.4	24.4	43.0
<b>Load <math>\mu</math>ops</b>	30.1	77.0	33.4	84.2
<b>Load L1 hits</b>	24.3	75.9	28.8	83.1
<b>Load L2 hits</b>	1.1	0.05	1.2	0.1
<b>Load L3 hits</b>	0.9	0.0	0.5	0.0
<b>Load L3 misses</b>	0.3	0.1	0.3	0.3

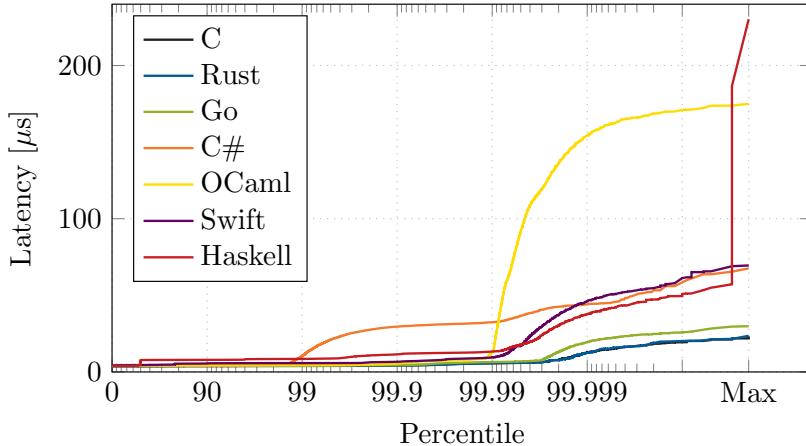
**Table 6:** Performance counter readings in events per packet when forwarding packets

# Why is Rust slower than C?

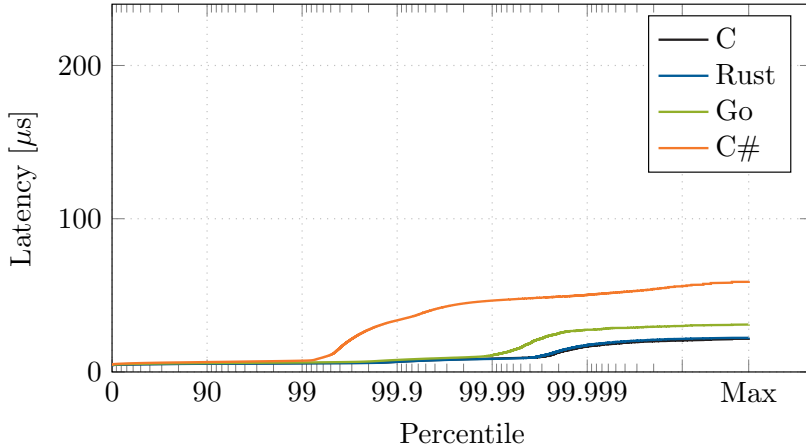
Events per packet	Batch 32, 1.6 GHz		Batch 8, 1.6 GHz	
	C	Rust	C	Rust
<b>Cycles</b>	94	100	108	120
<b>Instructions</b>	127	209	139	232
<b>Instr. per cycle</b>	1.35	2.09	1.29	1.93
<b>Branches</b>	18	24	19	27
<b>Branch mispredicts</b>	0.05	0.08	0.02	0.06
<b>Store <math>\mu</math>ops</b>	21.8	37.4	24.4	43.0
<b>Load <math>\mu</math>ops</b>	30.1	77.0	33.4	84.2
<b>Load L1 hits</b>	24.3	75.9	28.8	83.1
<b>Load L2 hits</b>	1.1	0.05	1.2	0.1
<b>Load L3 hits</b>	0.9	0.0	0.5	0.0
<b>Load L3 misses</b>	0.3	0.1	0.3	0.3

**Table 7:** Performance counter readings in events per packet when forwarding packets

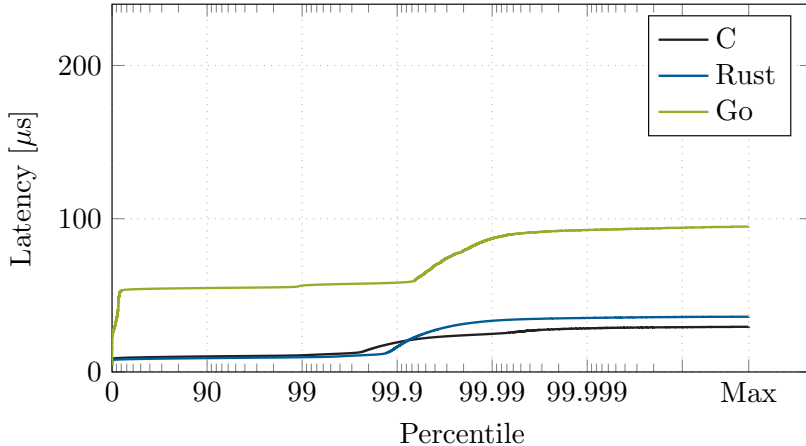
## Tail latency at 1 Mpps



## Tail latency at 10 Mpps



## Tail latency at 20 Mpps



# Unprivileged user space drivers

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

# Unprivileged user space drivers

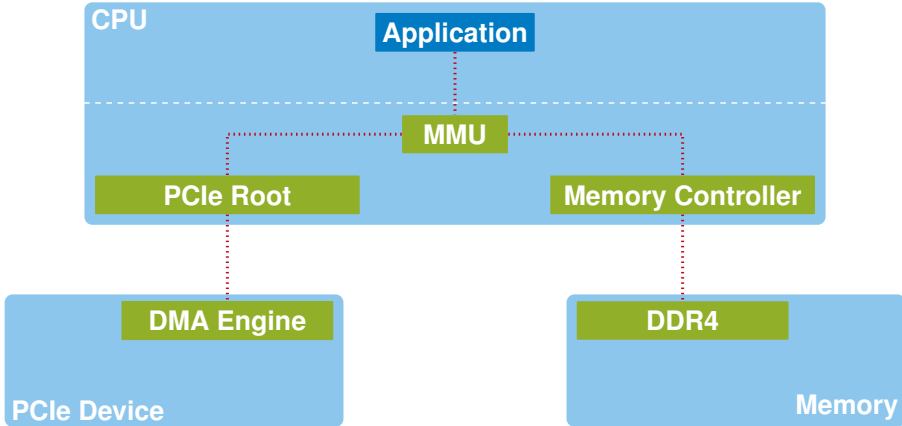
- User space drivers usually run with root privileges, but why?
- Mapping PCIe resources requires root
- Allocating non-transparent huge pages requires root (weird implementation detail)
- Locking memory requires root
- Can we do that in a small separate program that is easy to audit and then drop privileges?



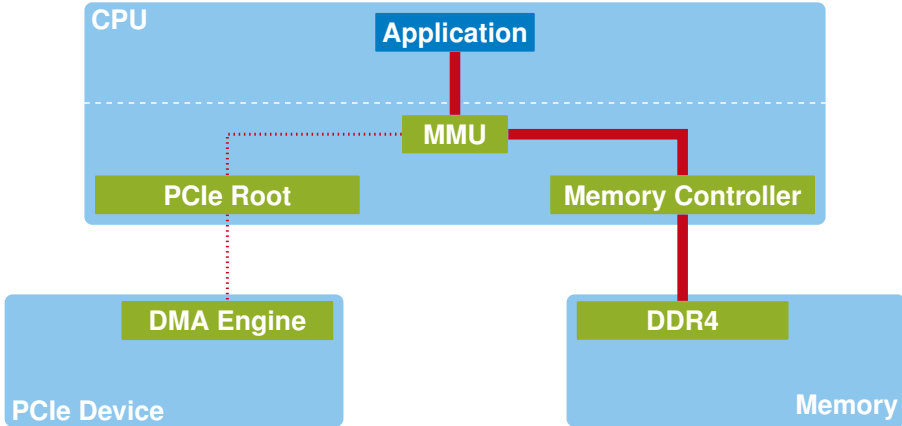
# Unprivileged user space drivers

- User space drivers usually run with root privileges, but why?
- Mapping PCIe resources requires root
- Allocating non-transparent huge pages requires root (weird implementation detail)
- 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

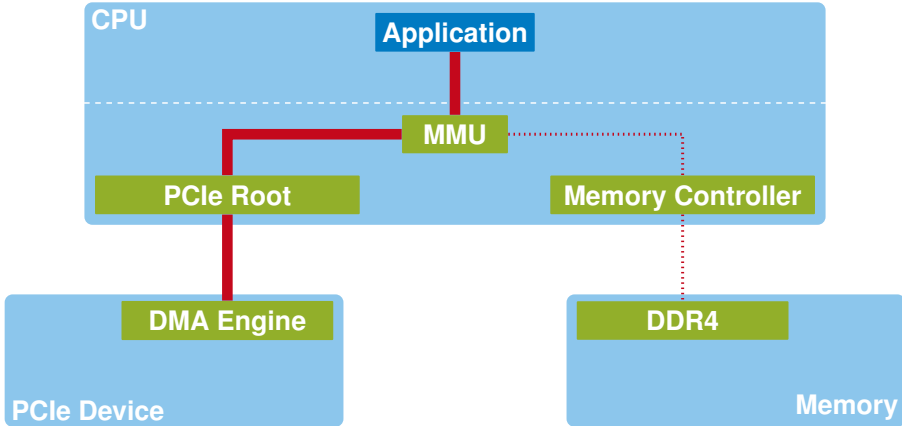
# Memory access on modern systems



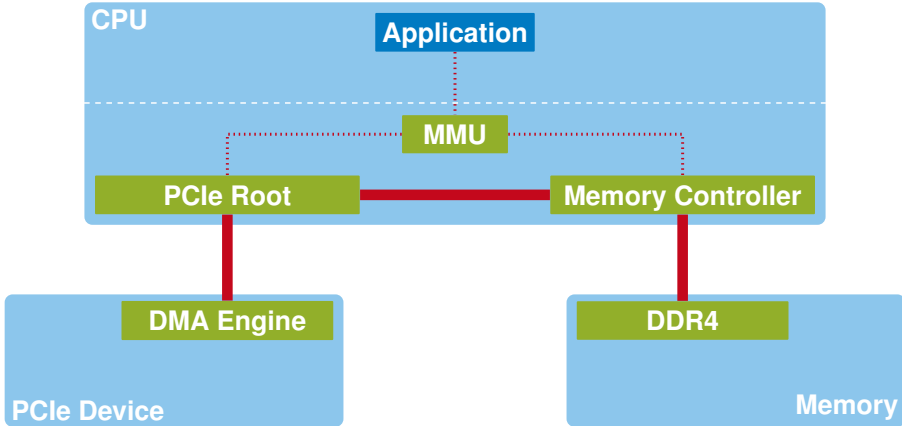
# Memory access on modern systems



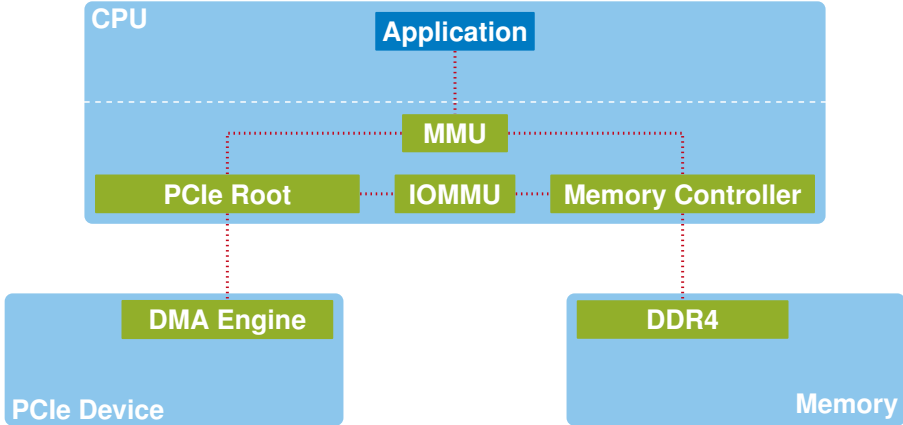
# Memory access on modern systems



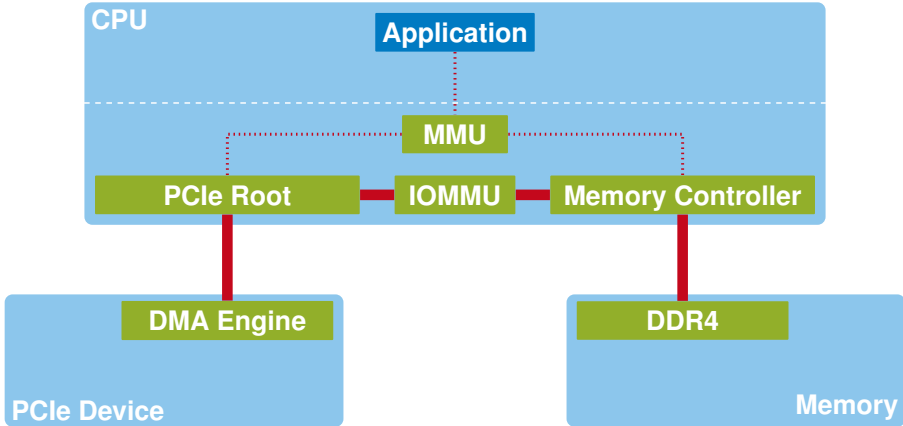
# Memory access on modern systems



# Memory access on modern systems: IOMMU to the rescue



# Memory access on modern systems: IOMMU to the rescue



# 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`



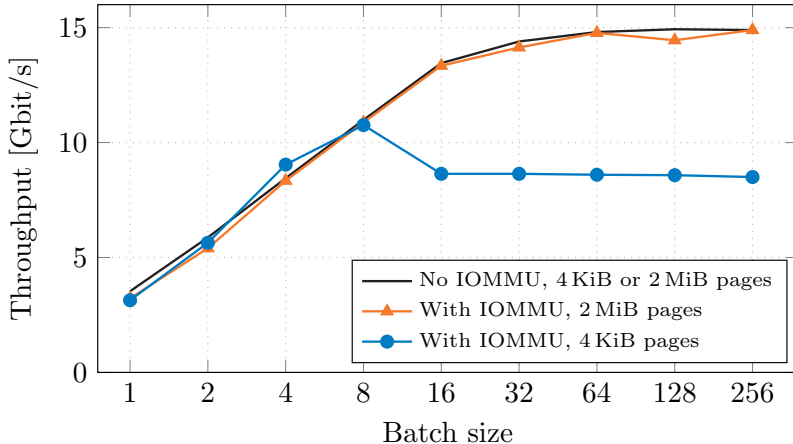
# Unprivileged user space drivers on Linux

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

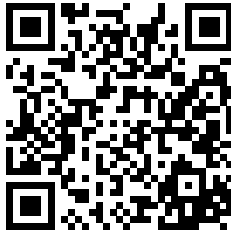
# Unprivileged user space drivers on Linux

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 C and Rust

## Does the IOMMU cost performance?



## 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
- Should your driver really be in the kernel?
- Next time you write a driver: consider a user space driver in a cool language