The Case for Writing Network Drivers in High-Level Programming Languages

Paul Emmerich[1], Simon Ellmann[1], Fabian Bonk[1,*], Alex Egger[1,*], Esaú García Sánchez-Torija[1,2,*], Thomas Günzel[1,*], Sebastian Di Luzio[1,*], Alexandru Obada[1,*], Maximilian Stadlmeier[1,3,*], Sebastian Voit[1,*], Georg Carle[1]

[1] Technical University of Munich
[2] Universitat Politècnica de Catalunya
[3] Ludwig-Maximilians-Universität München
{emmericp | ellmann | bonkf | eggera | garciasa | guenzel | luzio | aobada | mstadlme | voit | carle}@net.in.tum.de

ABSTRACT

Drivers are written in C or restricted subsets of C++ on all production-grade server, desktop, and mobile operating systems. They account for 66% of the code in Linux, but 39 out of 40 security bugs related to memory safety found in Linux in 2017 are located in drivers. These bugs could have been prevented by using high-level languages for drivers. We present user space drivers for the Intel ixgbe 10 Gbit/s network cards implemented in Rust, Go, C#, Java, OCaml, Haskell, Swift, JavaScript, and Python written from scratch in idiomatic style for the respective languages. We quantify costs and benefits of using these languages: High-level languages are safer (fewer bugs, more safety checks), but run-time safety checks reduce throughput and garbage collection leads to latency spikes. Out-of-order CPUs mitigate the cost of safety checks: Our Rust driver executes 63% more instructions per packet but is only 4% slower than a reference C implementation. Go's garbage collector keeps latencies below 100 μs even under heavy load. Other languages fare worse, but their unique properties make for an interesting case study.

All implementations are available as free and open source at https://github.com/ixy-languages/ixy-languages.

KEYWORDS

Rust, Go, C#, Java, OCaml, Haskell, Swift, JavaScript, Python

1 INTRODUCTION

C has been the go-to language for writing kernels since its inception. Device drivers are also mainly written in C, or restricted subsets of C++ providing barely no additional safety features, simply because they are tightly coupled with the kernel in all mainstream operating systems. Network device drivers managed to escape the grasp of the kernel with user space drivers such as DPDK [40] in the last years. Yet, all drivers in DPDK are written in C as large parts of them are derived from kernel implementations. DPDK consists of more than drivers: it is a whole framework for building fast packet processing apps featuring code from utility functions to complex data structures — and everything is written in C. This is not an unreasonable choice: C offers all features required for low-level systems programming and allows fine-grained control over the hardware to achieve high performance.

But with great power comes great responsibility: writing safe C code requires experience and skill. It is easy to make subtle mistakes that introduce bugs or even security vulnerabilities. Some of these bugs can be prevented by using a language that enforces certain safety properties of the program. Our research questions are: Which languages are suitable for driver development? What are the costs of safety features? A secondary goal is to simplify driver prototyping and development by providing the primitives required for user space drivers in multiple high-level languages.

We implement a user space driver tuned for performance for the Intel ixgbe family of network controllers (82599ES, X540, X550, and X552) in 9 different high-level languages featuring all major programming paradigms, memory management modes, and compilation techniques. All implementations are written from scratch in idiomatic style for the language by experienced programmers in the respective language and follow the same basic architecture, allowing for a performance comparison between the high-level languages and a reference C implementation. For most languages our

[*] Equal contribution, ordered alphabetically

ANCS'19, 2019

Language	Main paradigm*	Memory mgmt.	Compilation
Rust	Imperative	Ownership/RAII†	Compiled‡
Go	Imperative	Garbage collection	Compiled
C#	Object-oriented	Garbage collection	JIT
Java	Object-oriented	Garbage collection	JIT
OCaml	Functional	Garbage collection	Compiled
Haskell	Functional	Garbage collection	Compiled‡
Swift	Protocol-oriented [1]	Reference counting	Compiled‡
JavaScript	Imperative	Garbage collection	JIT
Python	Imperative	Garbage collection	Interpreted

* All selected languages are multi-paradigm

† Resource Acquisition Is Initialization

‡ Using LLVM

Table 1: Languages used by our implementations

driver is the first PCIe driver implementation tuned for performance enabling us to quantify the costs of different language safety features in a wide range of high-level languages. Table 1 lists the core properties of the selected languages.

2 RELATED WORK

Operating systems and device drivers predate the C programming language (1972 [64]). Operating systems before C were written in languages on an even lower level: assembly or ancient versions of ALGOL and Fortran. Even Unix started out in assembly language in 1969 before it was re-written in C in 1973 [65]. C is a high-level language compared to the previous systems: it allowed Unix to become the first portable operating system running on different architectures in 1977 [68]. The first operating systems in a language resembling a modern high-level language were the Lisp machines in the 70s (fueled by an AI hype). They featured operating systems written in Lisp that were fast due to hardware acceleration for high-level language constructs such as garbage collection. Both the specialized CPUs and operating systems died in the AI winter in the 80s [12]. Operating systems development has been mostly stuck with C since then.

Contemporary related work can be split into three categories: (1) operating systems and unikernels in high-level languages, (2) packet processing frameworks and user space drivers, and (3) language wrappers making the former available to applications in high-level languages. Operating systems, unikernels, and language wrappers are discussed with our implementations in the respective languages in Section 4.

Network functions with high performance requirements moved from the kernel into dedicated user space applications in the last decade. This move happened in two steps: first, frameworks like PF_RING DNA (2010) [48], PSIO (2010) [26, 27], netmap (2011) [67], and PFQ (2012) [8] provided a fast-path to the network driver from user space applications. They speed up packet IO by providing a kernel module that maps DMA buffers into a user space application. These frameworks are not user space drivers: all of them rely on a driver running in the kernel. The next step were true user space drivers: DPDK (open sourced in 2013) and Snabb (2012) move the entire driver logic into a user space process by mapping PCIe resources and DMA buffers into a user space library, running the driver in the same process as the application.

An example for this trend is the Click modular router [38] that started out as a kernel extension in 1999 and was later sped up with a netmap interface in 2012 as a demonstration of netmap itself [67]. Finally, a DPDK backend was added in 2015 to increase performance even further [7]. Similar migration paths can also be found in other open source projects: Open vSwitch comes with a kernel module and had plans to add both netmap and DPDK [57], the DPDK backend was merged in 2014, the netmap version never left the proof of concept stage [52]. pfSense [58] experimented with both netmap and DPDK in 2015 and finally chose DPDK [34]

Applications are moving to user space drivers in the form of DPDK and are therefore free of restrictions imposed by the kernel environment. Yet, all drivers in DPDK are still written in C. Snabb [42] (less popular than DPDK and only 4 drivers vs. DPDK's 27 drivers) is the only other performance-optimized user space driver not written in C: It comes with drivers written in Lua running in LuaJIT [54]. However, it makes extensive use of the LuaJIT foreign function interface [55] that erodes memory safety checks that are usually present in Lua. We are not including Snabb in our performance comparison because its architecture requires ring buffers to connect drivers and "apps", this makes it significantly slower than our drivers for the evaluated use case.

Unrelated work: Orthogonal to our proposal is the work on XDP [32] as it does not replace drivers but adds a faster interface on top of them. Moreover, eBPF code for XDP is usually written in a subset of C. P4 [9] also deserves a mention here, it is a high-level language for packet processing (but not for the driver itself). It primarily targets hardware, software implementations run on top of existing C drivers.

3 MOTIVATION

Proponents of operating systems written in high-level languages such as Biscuit (Go) [13], Singularity (Sing#, related to C#) [29], JavaOS [66], House (Haskell) [25], and Redox (Rust) [62] tout their safety benefits over traditional implementations. Of these only Redox is under active development with the goal of becoming a production system, the others are research projects and/or abandoned. These systems are safer, but it is unlikely that the predominant desktop and server operating systems will be replaced any time soon.

We argue that it is not necessary to replace the entire operating system at once. It is feasible to start writing user space drivers in safer languages today, gradually moving parts of the system to better languages. Drivers are also the largest attack surface (by lines of code) in modern operating

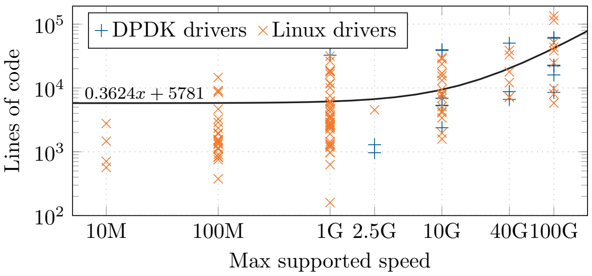


Figure 1: NIC technology node vs. driver size

systems and they keep growing in complexity as more and more features are added. There are real-world security issues in drivers that could have been prevented if they were written in a high-level language. Moving to user space drivers also means moving to a microkernel architecture with an IPC mechanism between drivers and applications. User space drivers are more isolated from the rest of the system than kernel drivers and can even run without root privileges if IOMMU hardware is available [17, 30].

3.1 Growing Complexity of Drivers

66% of the code in Linux 4.19 (current LTS) is driver code (11.2 M lines out of 17 M in total), 21% (2.35 M lines) of the driver code is in network drivers. 10 years ago in 2009 Linux 2.6.29 had only 53% of the code in drivers (3.7 M out of 6.9 M lines). Going back 10 more years to Linux 2.2.0 in 1999, we count 54% in drivers (646 k out of 1.2 M) with only 13% in network drivers. One reason for the growing total driver size is that there are more drivers to support more hardware.

Individual drivers are also increasing in complexity as hardware complexity is on the rise. Many new hardware features need support from drivers, increasing complexity and attack surface even when the number of drivers running on a given system does not change. Figure 1 shows a linear correlation ($R^2 = 0.3613$) between NIC technology node and driver complexity as a log-log scatter plot. The plot considers all Ethernet drivers in Linux 4.19 and all network drivers in DPDK that do not rely on further external libraries. We also omit DPDK drivers for FPGA boards (as these expect the user to bring their own hardware and driver support), unfinished drivers (Intel ice), drivers for virtual NICs, and 4 obsolete Linux drivers for which we could not identify the speed. The linear relationship implies that driver complexity grows exponentially as network speed increases exponentially.

3.2 Security Bugs in Operating Systems

Cutler et al. evaluate security bugs leading to arbitrary code execution in the Linux kernel found in 2017 [13]. They identify 65 security bugs leading to code execution and find that 8 of them are use-after-free or double-free bugs, 32 are out-of-bounds accesses, 14 are logic bugs that cannot be prevented by the programming language, and 11 are bugs where the effect of the programming language remains unclear [13]. The 40 memory bugs (61% of the 65 bugs) can be mitigated by using a memory-safe language such as their Go operating system Biscuit or our implementations. Performing an out-of-bounds access is still a bug in a memory-safe language, but it will only crash the program if it remains unhandled; effectively downgrading a code execution vulnerability to a denial of service vulnerability. User space drivers can simply be restarted after a crash, crashing kernel drivers usually take down the whole system.

We analyze these 40 memory bugs identified by Cutler et al. further and find that 39 of them are located in 11 different device drivers (the other bug is in the Bluetooth stack). The driver with the most bugs in this data set is the Qualcomm WiFi driver with 13 bugs, i.e., a total of 20% of all code execution vulnerabilities in the study could have been prevented if this network driver was written in a memory safe high-level language. The key result here is that rewriting device drivers in safer languages achieves 97.5% of the improvement they gained by rewriting the whole operating system.

Year	DPDK*	netmap†	Snabb*	PF_RING†	PSIO†	PFQ†
2010	0/0	0/0	0/0	0/0	1/1	0/0
2011	0/0	0/0	0/0	1/1	0/0	0/0
2012	0/1	1/4	0/0	0/4	0/1	0/1
2013	0/0	3/8	0/0	0/0	0/1	0/0
2014	4/11	4/14	0/0	0/4	1/1	1/2
2015	9/15	6/14	1/2	4/7	2/3	1/2
2016	12/22	1/12	0/1	0/1	0/0	1/1
2017	17/23	3/10	0/0	1/2	1/1	1/1
Sum	41/72	19/64	1/3	6/19	5/8	4/7

* User space driver
† Kernel driver with dedicated user space API

Table 2: Packet processing frameworks used in academia, cells are uses/mentions; e.g., 1/3 means 3 papers mention the framework, 1 of them uses it

3.3 The Rise of DPDK

Section 2 used the open source projects Click, Open vSwitch, and pfSense as examples to show how network applications moved from running completely in the kernel to user space frameworks with a kernel driver (e.g., netmap) to full user space drivers (e.g., DPDK). This trend towards DPDK is also present in academia. We run a full-text search for all user space packet processing frameworks on all publications between 2010 and 2017 in the ACM conferences SIGCOMM and ANCS and the USENIX conferences NSDI, OSDI, and ATC ($n = 1615$ papers). This yields 113 papers which we skimmed to identify whether they merely mention a framework or whether they build their application on top of it or even present the framework itself. Table 2 shows how DPDK started to dominate after it was open sourced in 2013[1].

3.4 Languages Used for DPDK Applications

Applications on top of DPDK are also not restricted to C, yet most users opt to use C when building a DPDK application. This is a reasonable choice given that DPDK comes with C example code and C APIs, but there is no technical reason preventing programmers from using any other language. In fact, DPDK applications in Rust, Go, and Lua exist [16, 31, 56].

DPDK's website showcases 21 projects building on DPDK. 14 (67%) of these are written in C, 5 (24%) in C++, 1 in Lua [16], and 1 in Go [31]. There are 116 projects using the #dpdk topic on GitHub, 99 of these are applications using DPDK; the others are orchestration tools or helper scripts for DPDK setups. 69 (70%) of these are written in C, 12 in C++, 4 in Rust (3 related to NetBricks [56], 1 wrapper), 4 in Lua (all related to MoonGen [16]), 3 in Go (all related to NFF-Go [31]), and one in Crystal (a wrapper).

Determining whether 67% to 70% of applications being written in C is unusually high requires a baseline to compare against. Further up the network stack are web application frameworks (e.g., nodejs or jetty) that also need to provide high-speed primitives for applications on the next higher layer. Of the 43 web platforms evaluated in the TechEmpower benchmark [71] (the largest benchmark collection featuring all relevant web platforms) only 3 (7%) are written in C. 17 different languages are used here, the dominant being Java with 19 (44%) entries. Of course not all platforms measured here are suitable for applications requiring high performance. But even out of the 20 fastest benchmarked applications ("Single query" benchmark) only one is written in C and 3 in C++. Java still dominates with 7 here. Go (3), Rust (2), and C# (1) are also present in the top 20. This shows that it is possible to build fast applications on a higher layer in languages selected here.

3.5 User Study: Mistakes in DPDK Applications Written in C

We task students with implementing a simplified IPv4 router in C on top of DPDK for one of our networking classes. The students are a mix of undergraduate and postgraduate students with prior programming experience, a basic networking class is a pre-requisite for the class. Students are provided with a skeleton layer 2 forwarding program that handles DPDK setup, threading logic, and contains a dummy routing table. Only packet validation according to RFC 1812 [5], forwarding via a provided dummy routing table implementation, and handling ARP is required for full credits. ICMP and fragmentation handling is not required.

Total	No mistakes	Logic error	Use-after-free	Int overflow
55	12	28	13	14
100%	22%	51%	24%	25%

Table 3: Mistakes made by students when implementing an IPv4 router in C on top of DPDK

We received 55 submissions with at least partially working code, i.e., code that at least forwards some packets, in which we identified 3 types of common mistakes summarized in Table 3. Incorrect programs can contain more than one class of mistake. Logic errors are the most common and include failing to check EtherTypes, forgetting validations, and getting checksums wrong, these cannot be prevented by safer languages. Memory bugs including use-after-free bugs can be prevented by the language. No out-of-bounds access was made because the exercise offers no opportunity to do so: the code only needs to operate on the fixed-size headers which are smaller than the minimum packet size. All integer overflow bugs happened due to code trying to decrement the time-to-live field in place in the packet struct without first checking if the field was already 0.

Ethical considerations. As handling errors done by humans requires special care we took all precautions to protect the privacy of the involved students. The study was conducted by the original corrector of the exercise, these results are used for teaching the class. No student code, answers, or any identifying information was ever given to anyone not involved in teaching the class. All submissions are pseudonymized. We were able to achieve the ethical goals of avoiding correlating errors with persons.

3.6 Summary

To summarize: (network) drivers are written in C which leads to preventable bugs in both the drivers (Sec. 3.2) and in applications on top of them (Sec. 3.5). Historical reasons for writing them in C no longer apply (Sec. 3.3). Driver complexity is growing (Sec. 3.1), so let's start using safer languages.

4 IMPLEMENTATIONS

All of our implementations are written from scratch by experienced programmers in idiomatic style for the respective language. We target Linux on x86 using the `uio` subsystem to map PCIe resources into user space programs. For details on implementing user space drivers, have a look at our original publication about the C driver [17].

4.1 Architecture

All of our drivers implement the same basic architecture as our ixy user space driver [17] which serves as reference

[1] The paper mentioning DPDK in 2012 is the original netmap paper [67] referring to DPDK as a commercial offering with similar goals

implementation that our high-level drivers are compared against. Ixy's architecture is inspired by DPDK, i.e., it uses poll mode drivers without interrupts, all APIs are based on batches of packets, and DMA buffers are managed in custom memory pools. The memory pool for DMA buffers is also needed despite automatic memory management: not all memory is suitable for use as DMA buffers, simply using the language's allocator is not possible. See [17] for details on DMA allocation for user space drivers. This restriction potentially circumvents some of the memory safety properties of the languages in parts of the driver.

It is a common misconception that user space drivers do not support interrupts in general. The Linux `vfio` subsystem offers full support for interrupts in user space drivers via an event file descriptor allowing handling interrupts with normal IO syscalls [41] that are available in high-level languages. However, interrupts lead to packet loss for packet rates above 2 Mpps on our test systems in a C implementation. They are useful as power-saving mechanism under low load and should be supported in a real-world driver, but we did not implement support for them in most drivers as we are mainly interested in peak performance.

4.2 Challenges for High-Level Languages

There are three main challenges for user space drivers in high-level languages compared to kernel drivers in C.

4.2.1 Handling external memory. Two memory areas cannot be managed by the language itself: Memory-mapped IO regions are provided by the device and DMA buffers must be allocated with a custom allocator. Languages need access to the `mmap` and `mlock` syscalls to allocate these memory areas. We use a small C function in languages where these syscalls are either not available at all or only supported with restricted flags.

4.2.2 Unsafe primitives. External memory, i.e., PCIe address space and DMA buffers, must be wrapped in language-specific constructs that enforce bounds checks and allow access to the backing memory. Many languages come with dedicated wrapper structs that are constructed from a raw pointer and a length. For other languages we have to implement these wrappers ourselves.

In any case, all drivers need to perform inherently unsafe memory operations that cannot be checked by any language feature. The goal is to restrict these unsafe operations to as few places as possible to reduce the amount of code that needs to be manually audited for memory errors.

4.2.3 Memory access semantics. Memory-mapped IO regions are memory addresses that are not backed by memory, each access is forwarded to a device and handled there. Simply telling the language to read or write a memory location in these regions can cause problems as optimizers make assumptions about the behavior of the memory. For example, writing a control register and never reading it back looks like a dead store to the language and the optimizer is free to remove the access. Repeatedly reading the same register in a loop while waiting for a value to be changed by the device looks like an opportunity to hoist the read out of the loop.

C solves this problem with the `volatile` keyword guaranteeing that at least one read or write access is performed. The high-level language needs to offer control over how these memory accesses are performed. Atomic accesses and memory barriers found in concurrency utilities make stronger guarantees and can be used instead if the language does not offer volatile semantics. Primitives from concurrency utilities can also substitute the access-exactly-once semantics required for some device drivers.

Readers interested in gory details about memory semantics for device drivers are referred to the Linux kernel documentation on memory barriers [39]. It is worth noting that all modern CPU architectures offer a memory model with cache-coherent DMA simplifying memory handling in drivers. We only test on x86 as proof of concept, but DPDK's support for x86, ARM, POWER, and Tilera shows that user space drivers themselves are possible on a wide range of modern architectures. Some of our implementations in dynamic languages likely rely on the strong memory model of x86 and might require modifications to work reliably on architectures with a weaker memory model such as ARM.

4.3 Rust Implementation

Rust is an obvious choice for safe user space drivers: safety and low-level features are two of its main selling points. Its ownership-based memory management allows us to use the native memory management even for our custom DMA buffers. We allocate a lightweight Rust struct for each packet that contains metadata and owns the raw memory. This struct is essentially being used as a smart pointer, i.e., it is often stack-allocated. This object is in turn either owned by the memory pool itself, the driver, or the user's application. The compiler enforces that the object has exactly one owner and that only the owner can access the object, this prevents use-after-free bugs despite using a completely custom allocator. Rust is the only language evaluated here that protects against use-after-free bugs and data races in memory buffers.

External memory is wrapped in `std::slice` objects that enforce bounds checks on each access, leaving only one place tagged as unsafe that can be the source of memory errors: we need to pass the correct length when creating the slice. Volatile memory semantics for accessing device registers are available in the `ptr` module.

Driver	NIC Speed	Code [Lines]	Unsafe [Lines]	% Unsafe
Our Linux implementation	10 Gbit/s	961	125	13.0%
Our Redox implementation	10 Gbit/s	901	68	7.5%
Redox e1000	1 Gbit/s	393	140	35.6%
Redox rtl8168	1 Gbit/s	363	144	39.8%

Table 4: Unsafe code in different Rust drivers

4.3.1 IOMMU and interrupt support. We also implemented support for `vfio` in the Rust driver, all other implementations only support the simpler `uio` interface. This interface enables us to use the IOMMU to isolate the device and run without root privileges and to use interrupts instead of polling under low load. Moreover, this interface offers a portable approach to allocating DMA memory in user space; the other implementations are specific to x86 and rely on an implementation detail in Linux (see [17]).

4.3.2 Related work. NetBricks (2016) [56] is a network function framework that allows users to build and compose network functions written in Rust. It builds on DPDK, i.e., the drivers it uses are written in C. They measure a performance penalty of 2% to 20% for Rust vs. C depending on the network function being executed.

We also ported our driver to Redox (2015) [62], a real-world operating system under active development featuring a microkernel written in Rust with user space drivers. It features two network drivers for the Intel e1000 NIC family (predecessor of the ixgbe family used here) and Realtek rtl8168 NICs. Table 4 compares how much unsafe code their drivers use compared to our implementations. Inspecting the pre-existing Redox drivers shows several places where unsafe code could be made safe with some more work as showcased by our Redox port. Line count for our Linux driver includes all logic to make user space drivers on Linux work, our Linux version therefore has more unsafe code than the Redox version which already comes with the necessary functionality. These line counts also show the relationship between NIC speed and driver complexity hold true even for minimal drivers in other systems.

4.4 Go Implementation

Go is a compiled systems programming language maintained by Google that is often used for distributed systems. Memory is managed by a garbage collector tuned for low latency. It achieved pause times in the low millisecond range in 2015 [63] and sub-millisecond pause times since 2016 [4].

External memory is wrapped in slices to provide bounds checks. Memory barriers and volatile semantics are indirectly provided by the atomic package which offers primitives with stronger guarantees than required.

4.4.1 Related work. Biscuit (2018) [13] is a research operating system written in Go that features a network driver for the same hardware as we are targeting here. Unlike all other research operating systems referenced here, they provide an explicit performance comparison with C. They observe GC pauses of up to 115 μs in their benchmarks and an overall performance of 85% to 95% of an equivalent C version. Unfortunately it does not offer a feasible way to benchmark only the driver in isolation for a comparison.

NFF-GO (2017) [31] is a network function framework allowing users to build and compose network functions in Go. It builds on DPDK, i.e., the drivers it uses are written in C. Google's Fuchsia [22] mobile operating system features a TCP stack written in Go on top of C drivers.

4.5 C# Implementation

C# is a versatile JIT-compiled and garbage-collected language offering both high-level features and systems programming features. Several methods for handling external memory are available, we implemented support for two of them to compare them. `Marshal` in `System.Runtime.InteropServices` offers wrappers and bounds-checked accessors for external memory. C# also offers a more direct way to work with raw memory: its unsafe mode enables language features similar to C, i.e., full support for pointers with no bounds checks and volatile memory access semantics.

4.5.1 Related work. The Singularity (2004) research operating system [29] is written in Sing#, a dialect of C# with added contracts and safety features developed for use in Singularity. It comes with a driver for Intel 8254x PCI NICs that are architecturally similar to the 82599 NICs used here: their DMA ring buffers are virtually identical. All memory accesses in their drivers are facilitated by safe APIs offered by the Singularity kernel using contracts to ensure that the driver cannot access memory it is not supposed to access.

4.6 Java Implementation

Java is a JIT-compiled and garbage-collected language similar to C# (which was heavily inspired by Java). The only standardized way to access external memory is by calling into C using JNI, a verbose and slow foreign function interface. We target OpenJDK 12 which offers a non-standard way to handle external memory via the `sun.misc.Unsafe` object that provides functions to read and write memory with volatile access semantics. We implement and compare both methods here. Java's low-level features are inferior compared to C#, the non-standard `Unsafe` object is cumbersome to use compared to C#'s unsafe mode with full pointer support. Moreover, Java does not support unsigned integer primitives requiring work-arounds as hardware often uses such types.

4.6.1 Related work. JavaOS (1996) [66] was a commercial operating system targeting embedded systems and thin clients written in Java, it was discontinued in 1999. Their device driver guide [69] contains the source code of a 100 Mbit/s network driver written in Java as an example. The driver implements an interface for network drivers and calls out to helper functions and wrapper provided by JavaOS for all low-level memory operations.

4.7 OCaml Implementation

OCaml is a compiled functional language with garbage collection. We use OCaml `Bigarrays` backed by external memory for DMA buffers and PCIe resources, allocation is done via C helper functions. The `Cstruct` library [45] from the MirageOS project [43] allows us to access data in the arrays in a structured way by parsing definitions similar to C struct definitions and generating code for the necessary accessor functions.

4.7.1 Related work. We also ported our driver to MirageOS (2013) [43], a unikernel written in OCaml with the main goal of improving security. MirageOS is not optimized for performance (e.g., all packets are copied when being passed between driver and network stack) and no performance evaluation is given by its authors (performance regression tests are being worked on [46]). MirageOS targets Xen, KVM, and normal Unix processes. The Xen version has a driver-like interface for Xen netfront (not a PCIe driver, though), the KVM version builds on the Solo5 unikernel execution environment [14] that provides a VirtIO driver written in C. Our port is the first PCIe network driver written in OCaml in MirageOS, currently targeting mirage-unix as the other versions lack the necessary PCIe support.

4.8 Haskell Implementation

Haskell is a compiled functional language with garbage collection. All necessary low-level memory access functions are available via the `Foreign` package. Memory allocation and mapping is available via `System.Posix.Memory`.

4.8.1 Related work. House (2005) [25] is a research operating system written in Haskell focusing on safety and formal verification using P-Logic [35]. It provides a monadic interface to memory management, hardware, user-mode processes, and low-level device IO. No quantitative performance evaluation is given.

PFQ [8] is a packet processing framework offering a Haskell interface and pfq-lang, a specialized domain-specific language for packet processing in Haskell. It runs on top of a kernel driver in C. Despite the focus on Haskell it is mainly written in C as it relies on C kernel modules: PFQ is 75% C, 10% C++, 7% Haskell by lines of code.

4.9 Swift Implementation

Swift is a compiled language maintained by Apple mainly targeted at client-side application development. Memory in Swift is managed via automatic reference counting, i.e., the runtime keeps a reference count for each object and frees the object once it is no longer in use. Despite primarily targeting end-user applications, Swift also offers all features necessary to implement drivers. Memory is wrapped in `UnsafeBufferPointers` (and related classes) that are constructed from an address and a size. Swift only performs bounds checks in debug mode.

4.9.1 Related work. No other drivers or operating systems in Swift exist. The lowest level Swift projects that are available are the Vapor [72] and Kitura [37] frameworks for server-side Swift.

4.10 JavaScript Implementation

We build on Node.js [47] with the V8 JIT compiler and garbage collector, a common choice for server-side JavaScript. We use `ArrayBuffers` to wrap external memory in a safe way, these arrays can then be accessed as different integer types using `TypedArrays`, circumventing JavaScript's restriction to floating point numbers. We also use the `BigInt` type that is not yet standardized but already available in Node.js. Memory allocation and physical address translation is handled via a Node.js module in C.

4.10.1 Related work. JavaScript is rarely used for low-level code, the most OS-like projects are NodeOS [33] and OS.js [3]. NodeOS uses the Linux kernel with Node.js as user space. OS.js runs a window manager and applications in the browser and is backed by a server running Node.js on a normal OS. Neither of these implements driver-level code in JavaScript.

4.11 Python Implementation

Python is an interpreted scripting language with garbage collection. Our implementation uses Cython for handling memory (77 lines of code), the remainder of the driver is written in pure Python. Performance is not the primary goal of this version of our driver, it is the only implementation presented here that is not explicitly optimized for performance. It is meant as a simple prototyping environment for PCIe drivers and as an educational tool.

Writing drivers in scripting languages allows for quick turn-around times during development or even an explorative approach to understanding hardware devices in an interactive shell. We provide primitives for PCIe driver development in Python that we hope to be helpful to others as this is the first PCIe driver in Python to our knowledge.

Lang.	Lines of code[1]	Lines of C code[1]	Source size (gzip[2])
C [17]	831	831	12.9 kB
Rust	961	0	10.4 kB
Go	1640	0	20.6 kB
C#	1266	34	13.1 kB
Java	2885	188	31.8 kB
OCaml	1177	28	12.3 kB
Haskell	1001	0	9.6 kB
Swift	1506	0	15.9 kB
JavaScript	1004	262	13.0 kB
Python	1242	(Cython) 77	14.2 kB

[1] Incl. C code, excluding empty lines and comments, counted with `cloc`

[2] Compression level 6

Table 5: Size of our implementations stripped down to the core feature set

4.11.1 VirtIO driver. We also implemented a driver for virtual VirtIO [49] NICs here to make this driver accessible to users without dedicated hardware. A provided Vagrant [28] file allows spinning up a test VM to get started with PCIe driver development in Python in a safe environment.

4.11.2 Related work. Python is a popular [60] choice for user space USB drivers with the PyUSB library [61]. In contrast to our driver, it is mainly a wrapper for a C library. Python USB drivers are used for devices that either mainly rely on bulk transfers (handled by the underlying C library) or that do not require many transfers per second.

5 EVALUATION

Table 5 compares the code size as counted with `cloc` ignoring empty lines and comments, we also include the code size after compressing it with gzip to estimate information entropy as lines of code comparisons between different languages are not necessarily fair. We stripped features not present in all drivers (i.e., all (unit-)tests, alternate memory access implementations, VirtIO support in C and Python, IOMMU/interrupt support in C and Rust) for this evaluation. We also omit register definitions because several implementations contain automatically generated lists of > 1000 mostly unused constants for register offsets. All high-level languages require more lines than C, but the Rust, Haskell, and OCaml implementations are smaller in size as their formatting style leads to many short lines. Java and JavaScript require more C code due to boilerplate requirements of their foreign function interfaces.

Table 6 summarizes protections against classes of bugs available to both our driver implementations and applications built on top of them. The take-away here is that high-level languages do not necessarily increase the work-load for the implementor while gaining safety benefits. Subjectively, we have even found it easier to write driver code in high-level languages — even if more lines of code were required — after figuring out the necessary low-level details of the respective language (a one-time effort). We also discovered memory bugs in the original C implementation while porting the drivers.

	General memory		Packet buffers		
Lang.	OoB[1]	Use after free	OoB[1]	Use after free	Int overflows
C [17]	✗	✗	✗	✗	✗
Rust	✓	✓	(✓)[2]	✓	(✓)[5]
Go	✓	✓	(✓)[2]	(✓)[4]	✗
C#	✓	✓	(✓)[2]	(✓)[4]	(✓)[5]
Java	✓	✓	(✓)[2]	(✓)[4]	✗
OCaml	✓	✓	(✓)[2]	(✓)[4]	✗
Haskell	✓	✓	(✓)[2]	(✓)[4]	(✓)[6]
Swift	✓	✓	✗[3]	(✓)[4]	✓
JavaScript	✓	✓	(✓)[2]	(✓)[4]	(✓)[6]
Python	✓	✓	(✓)[2]	(✓)[4]	(✓)[6]

[1] Out of bounds accesses

[2] Bounds enforced by wrapper, constructor in unsafe or C code

[3] Bounds only enforced in debug mode

[4] Buffers are never free'd/gc'd, only returned to a memory pool

[5] Disabled by default

[6] Uses floating point or arbitrary precision integers by default

Table 6: Language-level protections against classes of bugs in our drivers and the C reference code

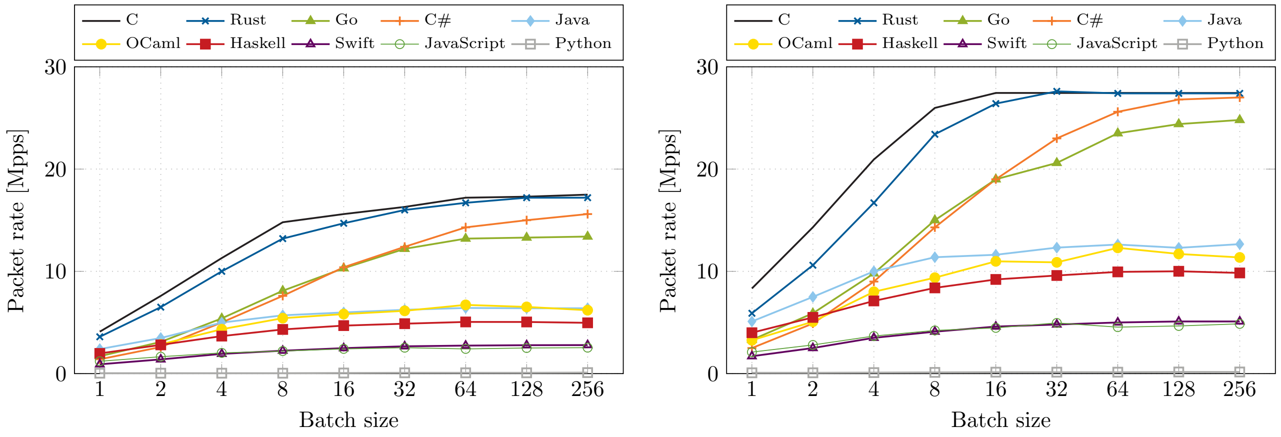
6 PERFORMANCE

The limiting factor for network drivers is the number of packets per second, not the bandwidth. All tests therefore use minimum-sized packets (up to 29.76 Mpps at 20 Gbit/s). We also benchmark the reference C driver ixy [17] as baseline performance measurement. The C driver proved to be as fast as older versions of DPDK but slower than modern versions of DPDK that offer a heavily vectorized transmit and receive path in its ixgbe driver [17].

6.1 Test Setup

We run our drivers on a Xeon E3-1230 v2 CPU clocked at 3.3 GHz with two 10 Gbit/s Intel X520 NICs. Test traffic is generated with MoonGen [16]. All drivers configure the NICs with a queue size of 512 (evaluation of different queue sizes can be found in [17]) and run a forwarding application that modifies one byte in the packet headers.

All tests are restricted to a single CPU core even though most of our implementations feature multi-core support. We already hit hardware limits with only one core. Multi-core scaling for network applications can be done at the hardware level even if the language does not support multi-threading. The NIC can split up incoming packets by hashing the flow and deliver them to independent processes. This enables trivial multi-core scaling independent of language restrictions, e.g., Snabb [42] uses multiple processes to scale the single-threaded LuaJIT runtime to multiple cores.



(a) Forwarding rate on 1.6 GHz CPU

(b) Forwarding rate on 3.3 GHz CPU

Figure 2: Forwarding rate of our implementations with different batch sizes

6.2 Effect of Batch Sizes

Processing packets in batches is a core concept of all fast network drivers [20]. Each received or sent batch requires synchronization with the network card, larger batches therefore improve performance. Too large batch sizes fill up the CPU's caches so there are diminishing returns or even reduced performance. 32 to 128 packets per batch is the sweet spot for user space drivers [6, 20, 36].

Figure 2 shows the maximum achievable bidirectional forwarding performance of our implementations. We also run the benchmark at a reduced CPU frequency of 1.6 GHz as the performance of the C and Rust forwarders quickly hit some hardware bottleneck at 94% line rate at 3.3 GHz. A few trade-offs for the conflicting goals of writing idiomatic safe code and achieving a high-performance were evaluated for each language. Haskell and OCaml allocate a new list/array for each processed batch of packets while all other languages re-use arrays. Recycling arrays in these functional languages building on immutable data structures would not be idiomatic code, this is one of the reasons for their lower performance.

6.2.1 Rust. Rust achieves 90% to 98% of the baseline C performance. Our Redox port of the Rust driver only achieves 0.12 Mpps due to performance bottlenecks in Redox (high performance networking is not yet a goal of Redox). This is still a factor 150 improvement over the pre-existing Redox drivers due to support for asynchronous packet transmission.

6.2.2 Go. Go also fares well, proving that it is a suitable candidate for a systems programming language.

6.2.3 C#. The aforementioned utility functions from the `Marshal` class to handle memory proved to be too slow to achieve competitive performance. Rewriting the driver to use C# unsafe blocks and raw pointers in selected places improved performance by 40%. Synthetic benchmarks show a 50-60% overhead for the safer alternatives over raw pointers.

6.2.4 Java. Our implementation heavily relies on devirtualization, inlining, and dead-code elimination by the optimizer as it features several abstraction layers and indirections that are typical for idiomatic Java code. We used profiling to validate that all relevant optimization steps are performed, almost all CPU time is spent in the transmit and receive functions showing up as leaf functions despite calling into an abstraction layer for memory access.

We use OpenJDK 12 with the HotSpot JIT, the *Parallel* garbage collector, and the Unsafe object for memory access. Using JNI for memory access instead is several orders of magnitude slower. Using OpenJ9 as JIT reduces performance by 14%. These results are significantly slower than C# and show that C#'s low-level features are crucial for fast drivers. One reason is that C# features *value types* that avoid unnecessary heap allocations. We try to avoid object allocations by recycling packet buffers, but writing allocation-free code in idiomatic Java is virtually impossible, so there are some allocations. OpenJDK 12 features 7 different garbage collectors which have an impact on performance as shown in Table 7. Epsilon is a no-op implementation that never frees memory, leaking approximately 20 bytes per forwarded packet.

Batch\GC	CMS	Serial	Parallel	G1	ZGC	Shenandoah	Epsilon
4	9.8	10.0	10.0	7.8	9.4	8.5	10.0
32	12.3	12.4	12.3	9.3	11.5	10.8	12.2
256	12.6	12.4	12.3	9.7	11.6	11.2	12.7

Table 7: Performance of different Java garbage collectors in Mpps when forwarding packets at 3.3 GHz

6.2.5 OCaml. Enabling the Flambda [50] optimizations in OCaml 4.07.0 increases throughput by 9%. An interesting optimization is representing bit fields as separate 16 bit integers instead of 32 bit integers if possible: Larger integers are boxed in the OCaml runtime. This increases performance by 0.7% when applied to the status bits in the DMA descriptors.

Our MirageOS port achieves 3 Gbit/s TCP throughput using iperf and the Mirage TCP stack. The bottleneck is MirageOS as it lacks support for batching and requires several copy operations for each packet.

6.2.6 Haskell. Compiler (GHC 8.4.3) optimizations seem to do more harm than good in this workload. Increasing the optimization level in the default GHC backend from `O1` to `O2` reduces throughput by 11%. The data in the graph is based on the LLVM backend which is 3.5% faster than the default backend at `O1`. Enabling the threaded runtime in GHC decreases performance by 8% and causes the driver to lose packets even at loads below 1 Mpps due to regular GC pauses of several milliseconds.

6.2.7 Swift. Swift increments a reference counter for each object passed into a function and decreases it when leaving the function. This is done for every single packet as they are wrapped in Swift-native wrappers for bounds checks. There is no good way to disable this behavior for the wrapper objects while maintaining an idiomatic API for applications using the driver. A total of 76% of the CPU time is spent incrementing and decrementing reference counters. This is the only language runtime evaluated here that incurs a large cost even for objects that are never free'd.

6.2.8 JavaScript. We also compare different Node.js versions: 10 (V8 6.9, current LTS), 11 (V8 7.0), and 12 (V8 7.5), older versions are unsupported due to lack of `BigInt` support. Node 10 and 11 perform virtually identical, upgrading to 12 degrades performance by 13% as access to `TypedArrays` is slower in this version. Performance optimizations applied to reach the current level are related to access `TypedArrays` (which are faster than plain `DataViews`) and reducing usage of `BigInts`: especially constructing BigInts is slow and most values required can be represented in the double data type.

6.2.9 Python. Python only achieves 0.14 Mpps in the best case using the default CPython interpreter in version 3.7. Most time is spent in code related to making C structs available to higher layers. We are using constructs that are incompatible with the JIT compiler PyPy. This is the only implementation here not optimized for performance, we believe it is possible to increase throughput by an order of magnitude by re-cycling struct definitions. Despite this we are content with the Python implementation as the main goal was not a high throughput but a simple implementation in a scripting language for prototyping functionality.

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

Table 8: Performance counter readings in events per packet when forwarding packets

6.3 The Cost of Safety Features in Rust

Rust is our fastest implementation achieving more than 90% of the performance of C when constrained by available CPU resources. It is also the only high-level language without overhead for memory management here, making it an ideal candidate to investigate overhead further by profiling. There are only two major differences between the Rust and C implementations:

1. Rust enforces bounds checks while the C code contains no bounds checks (arguably idiomatic style for C).
2. C does not require a wrapper object for DMA buffers, it stores all necessary metadata directly in front of the packet data the same memory area as the DMA buffer.

However, the Rust wrapper objects can be stack-allocated and effectively replace the pointer used by C with a smart pointer, mitigating the locality penalty. The main performance disadvantage is therefore bounds checking.

We use CPU performance counters to profile our forwarder with two different batch sizes. Table 8 lists the results in events per forwarded packet. Rust requires 65% (67%) more instructions to forward a single packet at a batch size of 32 (8). The number of branches executed rises by 33% (42%), the number of loads even by 150% (180%). However, the Rust code only requires 6% (11%) more cycles per packet overall despite doing more work. Synthetic benchmarks can achieve an even lower overhead of bounds checking [18]. A modern superscalar out of order processor can effectively hide the overhead introduced by these safety checks: normal execution does not trigger bounds check violations, the processor is therefore able to correctly predict (branch mispredict rate is at 0.2% - 0.3%) and speculatively execute the correct path.[2]

[2] A good example of speculatively executing code in the presence of bounds checks is the Spectre v1 security vulnerability which exists due to this performance optimization in CPUs [59]. Note that user space drivers are not affected by this vulnerability as the control flow does not cross a trust boundary as everything runs in the same process.

The Rust code achieves about 2 instructions per cycle vs. about 1.3 instructions with the C code.

Caches also help with the additional required loads of bounds information: this workload achieves an L1 cache hit rate of 98.5% (98.7%). Note that the sum of cache hits and L3 misses is not equal to the number of load μops because some loads are executed immediately after the store, fetching the data from the store buffer before it even reaches the cache.

Another safety feature in Rust are integer overflow checks that catch a common mistake in our user study, see Section 3.5. Overflow checks are currently disabled by default in release mode in Rust and have to be explicitly enabled with a compile-time flag. Doing so decreases throughput by only 0.8% at batch size 8, no statistically significant deviation was measurable with larger batch sizes. Profiling shows that 9 additional instructions per packet are executed by the CPU, 8 of them are branches. Total branch mispredictions are unaffected, i.e., the branch check is always predicted correctly by the CPU. This is another instance of speculative execution in an out-of-order CPU hiding the cost of safety features.

6.4 Comparison with Other Language Benchmarks

Table 9 compares our performance results with the Computer Language Benchmarks Game (CLBG) [24], a popular more general performance comparison of different languages. We use the "fastest measurement at the largest workload" data set from 2019-07-21, summarized as geometric mean [19] over all benchmarks. Our results are for batch size 32 (realistic value for real-world applications, e.g., DPDK default) at 1.6 GHz CPU speed to enforce a CPU bottleneck.

This shows that especially dynamic and functional languages pay a performance penalty when being used for low-level code compared to the more general benchmark results. Our implementations (except Python) went through several rounds of optimization based on profiling results and extensive benchmarks. While there are certainly some missed opportunities for further minor optimization, we believe to be close to the optimum achievable performance for drivers in idiomatic code in these languages. Note that the reference benchmark is also probably not perfect. Go and C# perform even better at the low-level task of writing drivers than the general purpose benchmarks, showing how a language's performance characteristics depend on the use case. General-purpose benchmark results can be misleading when writing low-level code in high-level languages.

Bench.\Lang.	Rust	Go	C#	Java	OCaml	Haskell	Swift	JS	Py.
Our results	98%	81%	76%	38%	38%	30%	16%	16%	1%
CLBG [24]	117%	34%	73%	52%	80%	65%	64%	28%	4%

Table 9: Performance results normalized to C, i.e., 50% means it achieves half the speed of C

7 LATENCY

Latency is dominated by time spent in buffers, not by time spent handling a packet on the CPU. Our drivers forward packets in hundreds of cycles, i.e., within hundreds of nanoseconds. A driver with a lower throughput is therefore not automatically one with a higher latency while operating below its load limit. The main factors driving up latency are pauses due to garbage collection and the batch size. Note that the batch size parameter is only the maximum batch size, a driver operating below its limit will process smaller batches. Drivers running closer to their limits will handle larger batches and incur a larger latency. Our drivers run with ring sizes of 512 by default and configure the NIC to drop packets if the receive ring is full, a common setting to avoid buffer bloat.

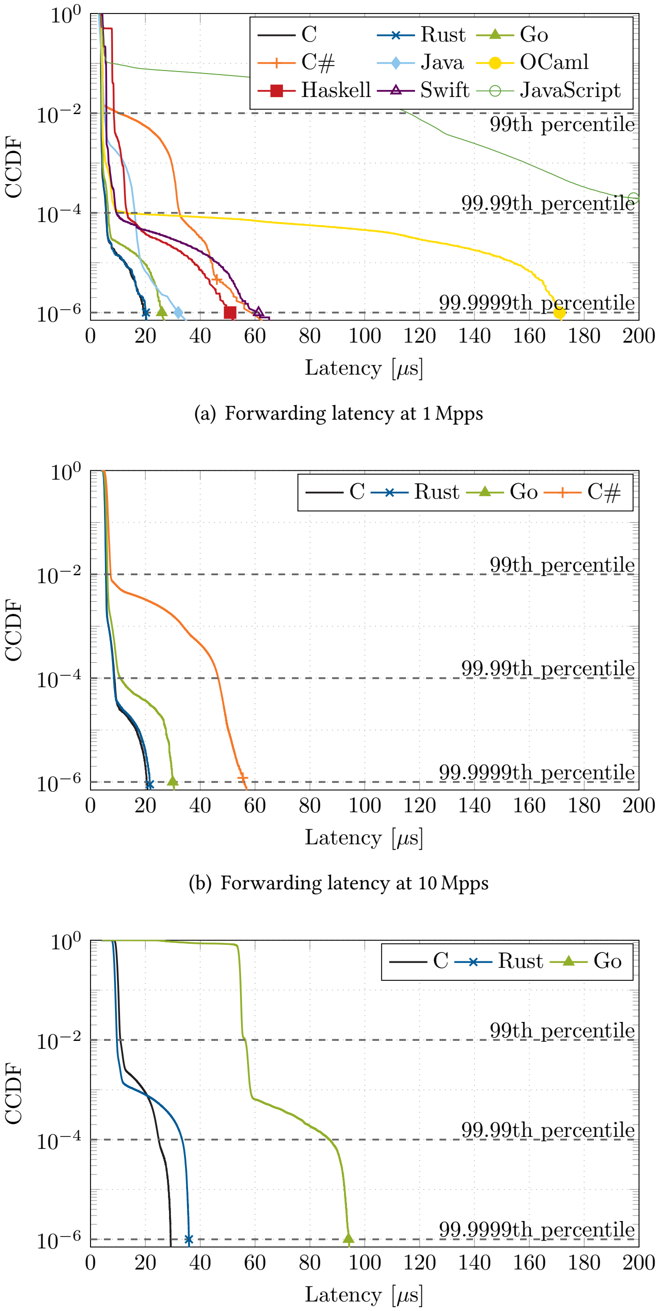
7.1 Test Setup

Choice of language does not affect the average or median latency significantly: garbage collection pauses and JIT compilation are visible in tail latency. We therefore measure the latency of all forwarded packets by inserting fiber optic splitters on both sides of the device under test. All packets are mirrored to a server running MoonSniff [2] with hardware timestamping on a Xeon D embedded NIC (measured precision 24 ns, captures all packets). The device under test uses an Intel Xeon E5-2620 v3 at 2.40 GHz and a dual-port Intel X520 NIC. All latencies were measured with a batch size of 32 and ring size 512 under bidirectional load with constant bit-rate traffic. The device under test has a maximum buffer capacity of 1,088 packets in this configuration. Different batch sizes, ring sizes, and NUMA configurations affect latency in the same way for all programming languages, interested readers are referred to the original publication about the C driver [17].

7.2 Tail Latencies

Figure 3 shows latencies of our drivers when forwarding packets at different rates. The data is plotted as CCDF to focus on the worst-case latencies. Implementations not able to cope with the offered load are omitted from the graphs — their latency is simply a function of the buffer size as the receive buffers fill up completely. No maximum observed latency is significantly different from the 99.9999th percentile. Java and JavaScript lose packets during startup due to JIT compilation, we therefore exclude the first 5 seconds of the test runs for these two languages. All other tests shown ran without packet loss.

7.2.1 Rust and C. Even Rust and C show a skewed latency distribution with some packets taking 5 times as long as the median packet. One reason for this is that all our drivers handle periodic (1 Hz) printing of throughput statistics in



(a) Forwarding latency at 1 Mpps

(b) Forwarding latency at 10 Mpps

(c) Forwarding latency at 20 Mpps

Figure 3: Tail latency of our implementations when forwarding packets

the main thread. Note that the 99.9999th percentile means that one in a million packets is affected. Printing statistics once per second at 1 Mpps or more thus affects latency at this level. A second reason is that it is not possible to isolate a core completely from the system on Linux. Some very short local timer interrupts are even present with the `isolcpus` kernel option. C outperforms Rust at 20 Mpps because Rust operates close to its limit of ≈24 Mpps on this system and processes larger batches.

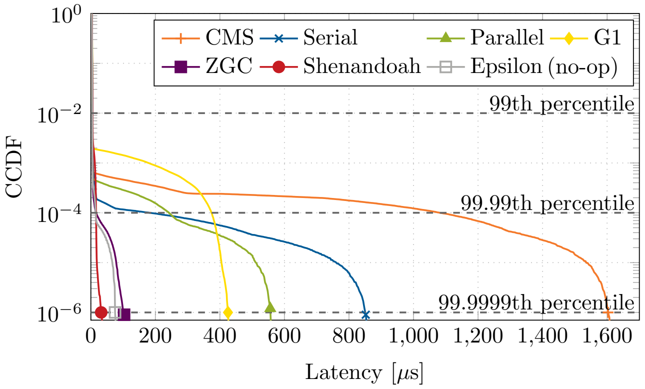


Figure 4: Forwarding latency of Java at 1 Mpps with different garbage collectors

7.2.2 *Go.*

Go's low-latency garbage collector achieves the lowest pause times of any garbage-collected language here. Latency suffers at 20 Mpps because the driver operates at its limit on this system here. Cutler et al. measured a maximum garbage collection pause of 115 μs in their Go operating system, demonstrating that sub-millisecond pauses with Go are possible even in larger applications.

7.2.3 *C#.*

C# features several garbage collector modes tuned for different workloads [44]. The default garbage collector caused a latency of 240 μs at the 99.9999th percentile at 10 Mpps. Switching it to `SustainedLowLatency` reduces latency to only 55 μs, this change also reduces the maximum achievable throughput by 1.2%. All measurements were performed with the `SustainedLowLatency` garbage collector.

C# also uses a JIT compiler that might introduce additional pause times. However, the compilation of most functions happens immediately after startup even if no packets are forwarded: We implement a poll-mode driver that effectively warms up the JIT compiler.

7.2.4 *Java.*

Java exhibits packet loss and excessive latencies during the first few seconds of all test runs, this is likely due to JIT compilation hitting a function only after the traffic flow starts. All latency measurements for Java therefore exclude the first 5 seconds of the test runs. We also disabled printing statistics for the latency test runs for the Shenandoah, Epsilon, and ZGC garbage collectors which reduced tail latency by ≈ 300 μs. The other garbage collectors perform *better* when statistics are printed. For example, G1's latency increases by ≈ 400 μs when it does not print statistics. No other language required disabling printing. Figure 3(a) shows results for the Shenandoah garbage collector which exhibited the lowest latency. We also tried the different settings in Shenandoah that are recommended for low-latency requirements [53] and found no significant improvements.

Figure 4 compares the latency incurred by the different available garbage collectors in OpenJDK 12 while forwarding 1 Mpps. We configured the lowest possible target pause time of 1 ms. Note that the maximum buffer time with this configuration is ≈1.1 ms, i.e., the CMS collector drops packets at this rate. This could be mitigated by larger rings or by enabling buffering on the NIC if the ring is full. There is a clear trade-off between throughput and latency for the different garbage collectors, cf. Table 7. ZGC hits a sweet spot between high throughput and low latency. We expect this to change with OpenJDK 13 which brings performance improvements for Shenandoah making it a good default in the future. Epsilon (no-op, never frees objects) is worse than Shenandoah: leaking memory leads to bad data locality as it fills up the whole address space.

7.2.5 OCaml and Haskell. Both OCaml and Haskell ship with only a relatively simple garbage collector (compared to Go, C#, and Java) not optimized for sub-millisecond pause times. Haskell even drops packets due to garbage collection pauses when the multi-threaded runtime is enabled, the single threaded runtime performs reasonably well. Haskell plans to add a new low-latency garbage collector [21].

7.2.6 Swift. It remains unclear why Swift performs worse than some garbage-collected languages. Its reference counting memory management should distribute the work evenly and not lead to spikes, but we observe tail latency comparable to the garbage-collected Haskell driver.

7.2.7 JavaScript. JavaScript loses packets during startup, indicating that the JIT compiler is to blame, the graph excludes the first 5 seconds. The latency is still off the chart, the 99.99th percentile is 261 μs, the 99.9999th percentile 353 μs and the maximum 359 μs.

7.2.8 Python. Python exhibits packet loss even at low rates and is therefore excluded here, worst-case latencies are several milliseconds even when running at 0.1 Mpps.

8 CONCLUSION

Rewriting the entire operating system in a high-level language is a laudable effort but unlikely to disrupt the big mainstream desktop and server operating systems in the near future. We propose to start rewriting drivers as user space drivers in high-level languages instead as they present the largest attack surface. 39 of the 40 memory safety bugs in Linux examined here are located in drivers, showing that most of the security improvements can be gained without replacing the whole operating system. Network drivers are a good starting point for this effort: User space network drivers written in C are already commonplace. Moreover, they are critical for security: they are exposed to the external world or serve as a barrier isolating untrusted virtual machines (e.g., CVE-2018-1059 in DPDK allowed VMs to extract host memory due to a missing bounds check [15]).

Higher layers of the network stack are also already moving towards high-level languages (e.g., the TCP stack in Fuchsia [22] is written in Go) and towards user space implementations. The transport protocol QUIC is only available as user space libraries, e.g., in Chromium [23] or CloudFlare's quiche written in Rust [11]. Apple runs a user space TCP stack on mobile devices [10]. User space stacks also call for a more modern interface than POSIX sockets: The socket replacement TAPS is currently being standardized, it explicitly targets "modern platforms and programming languages" [70]. This trend simplifies replacing kernel drivers with user space drivers as legacy interfaces are being deprecated.

Our evaluation shows that Rust is a prime candidate for safer drivers: its ownership-based management system prevents memory bugs even in custom memory areas not allocated by the language runtime. The cost of these safety and security features are only 2% - 10% of throughput on modern out-of-order CPUs. Rust's ownership based memory management provides more safety features than languages based on garbage collection here and it does so without affecting latency. Linux kernel developers recently discussed the possibility to accept Rust code in the kernel as an optional dependency to enable safer code [51].

Go and C# are also a suitable language if the system can cope with sub-millisecond latency spikes due to garbage collection. The other languages discussed here can also be useful if performance is less critical than having a safe and correct system, for example, Haskell and OCaml are more suitable for formal verification.

REPRODUCIBLE RESEARCH

We publish all of our source code and test scripts on GitHub at https://github.com/ixy-languages/ixy-languages. The only hardware requirement is a network card from the ixgbe family which are readily available from several vendors and often found as on-board NICs.

ACKNOWLEDGMENTS

This work was supported by the German-French Academy for the Industry of the Future. We would like to thank Andreas Molzer, Boris-Chengbiao Zhou, Johannes Naab, Maik Luu Bach, Olivier Coanet, Sebastian Gallenmüller, Stefan Huber, Stephan-Alexander Posselt, and Thomas Zwickl for valuable contributions to the drivers and/or this paper.

REFERENCES

[1] Dave Abrahams. 2015. Protocol-Oriented Programming in Swift. *WWDC15* (June 2015).

[2] Alexander Frank. 2018. MoonSniff. https://github.com/emmericp/MoonGen/pull/227. (2018).

[3] Anders Evenrud et al. 2019. OS.js. (2019). https://www.os-js.org.

[4] Austin Clements, Rick Hudson. 2016. Proposal: Eliminate STW stack re-scanning. (2016). https://go.googlesource.com/proposal/+/master/design/17503-eliminate-rescan.md.

[5] Fred Baker. 1995. RFC 1812: Requirements for IP version 4 routers. (1995).

[6] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. 2018. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine* 56, 12 (2018), 97–103.

[7] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *Architectures for Networking and Communications Systems (ANCS)*. ACM, 5–16.

[8] N. Bonelli, S. Giordano, and G. Procissi. 2016. Network Traffic Processing With PFQ. *IEEE Journal on Selected Areas in Communications* 34, 6 (June 2016), 1819–1833. https://doi.org/10.1109/JSAC.2016.2558998

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[10] Stuart Cheshire, David Schinazi, and Christoph Paasch. 2017. Advances in Networking. *WWDC17* (June 2017).

[11] Cloudflare. 2019. quice: Savoury implementation of the QUIC transport protocol. (2019). https://github.com/cloudflare/quiche.

[12] Daniel Crevier. 1993. *AI: The Tumultuous Search for Artificial Intelligence.* BasicBooks.

[13] Cody Cutler, M Frans Kaashoek, and Robert T Morris. 2018. The benefits and costs of writing a POSIX kernel in a high-level language. In *OSDI'18*. USENIX, 89–105.

[14] Dan Williams. 2019. Solo5 project. (2019). https://github.com/Solo5/solo5.

[15] DPDK Project. 2018. Vhost-user CVE-2018-1059. (2018). Mailing list post. http://mails.dpdk.org/archives/announce/2018-April/000192.html.

[16] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan.

[17] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. 2019. User Space Network Drivers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*.

[18] David Flater and William F Guthrie. 2013. A case study of performance degradation attributable to run-time bounds checks on C++ vector access. *Journal of research of the National Institute of Standards and Technology* 118 (2013), 260.

[19] Philip J Fleming and John J Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* 29, 3 (1986), 218–221.

[20] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. 2015. Comparison of Frameworks for High-Performance Packet IO. In *Architectures for Networking and Communications Systems (ANCS)*. ACM, Oakland, CA.

[21] Ben Gamari. 2018. A low-latency garbage collector for GHC. *MuniHac 2018* (2018).

[22] Google. 2019. Fuchsia git repositories. (2019). https://fuchsia.googlesource.com/.

[23] Google. 2019. Playing with QUIC. (2019). https://www.chromium.org/quic/playing-with-quic.

[24] Isaac Gouy. 2019. The Computer Language Benchmarks Game. (2019). https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[25] Thomas Hallgren, Mark P Jones, Rebekah Leslie, and Andrew Tolmach. 2005. A principled approach to operating system construction in Haskell. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 116–128.

[26] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: Packet I/O Engine. (2010). https://shader.kaist.edu/packetshader/io_engine/.

[27] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2011. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 195–206.

[28] HashiCorp. 2019. Vagrant website. (2019). https://www.vagrantup.com/.

[29] Galen Hunt, James R Larus, Martın Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. 2005. *An overview of the Singularity project.* Technical Report. MSR-TR-2005-135, Microsoft Research.

[30] Intel. 2018. Intel Virtualization Technology for Directed I/O. (2018). Rev. 3.0.

[31] Intel. 2019. Intel Virtualization Technology for Directed I/O. (2019). https://github.com/intel-go/nff-go.

[32] IO Visor Project. 2019. Introduction to XDP. (2019). https://www.iovisor.org/technology/xdp.

[33] Jesús Leganés-Combarro et al. 2019. Node-OS. (2019). https://node-os.com.

[34] Jim Thompson. 2017. DPDK, VPP & pfSense 3.0. In *DPDK Summit Userspace*.

[35] Richard B Kieburtz. 2002. P-logic: Property verification for Haskell programs. (2002).

[36] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. 2012. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*. ACM.

[37] Kitura project. 2019. Kitura website. (2019). https://www.kitura.io/.

[38] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[39] Linux documentation. 2019. Linux kernel memory barriers. (2019). https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[40] Linux Foundation. 2013. Data Plane Development Kit. (2013). http://dpdk.org.

[41] Linux Kernel Documentation. 2019. VFIO - Virtual Function I/O. (2019). https://www.kernel.org/doc/Documentation/vfio.txt.

[42] Luke Gorrie et al. 2012. Snabb: Simple and fast packet networking. (2012). https://github.com/snabbco/snabb.

[43] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[44] Microsoft. 2019. .NET Garbage Collection Latency Modes. (2019). https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/latency.

[45] MirageOS project. 2019. Cstruct. (2019). https://github.com/mirage/ocaml-cstruct.

[46] MirageOS project. 2019. Performance harness for MirageOS 3. (2019). https://github.com/mirage/mirage/issues/685.

[47] Node.js Foundation. 2019. Node.js. (2019). https://nodejs.org.

[48] ntop. 2010. Introducing PF_RING DNA (Direct NIC Access). (2010). https://www.ntop.org/pf_ring/introducing-pf_ring-dna-direct-nic-access/.

[49] OASIS VIRTIO TC. 2016. Virtual I/O Device (VIRTIO) Version 1.0. (March 2016). http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf.

[50] OCaml Manual. 2019. Optimisation with Flambda. (2019). https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html.

[51] Online discussion. 2019. Rust is the future of systems programming, C is the new Assembly. (Aug. 2019). https://lwn.net/Articles/797828/.

[52] Open vSwitch project. 2019. Open vSwitch releases. (2019). http://docs.openvswitch.org/en/latest/faq/releases/.

[53] OpenJDK. 2019. Shenandoah GC. (2019). https://wiki.openjdk.java.net/display/shenandoah/Main.

[54] Mike Pall. 2019. LuaJIT. http://luajit.org/. (2019).

[55] Mike Pall. 2019. LuaJIT FFI Library. http://luajit.org/ext_ffi.html. (2019).

[56] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI'16*. USENIX, 203–216.

[57] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX, Oakland, CA, 117–130.

[58] pfSense project. 2019. pfSense website. (2019). https://www.pfsense.org/.

[59] Project Zero. 2018. Meltdown and Spectre. (2018). https://meltdownattack.com/.

[60] PyPi. 2019. Download statistics for PyUSB. (2019). https://pypistats.org/packages/pyusb.

[61] PyPi. 2019. PyUSB package. (2019). https://pypi.org/project/pyusb/.

[62] Redox developers. 2019. Redox project page. (2019). https://www.redox-os.org/.

[63] Rick Hudson. 2015. Go GC: Latency Problem Solved. *GopherCon Denver* (July 2015).

[64] Dennis M Ritchie. 1993. The development of the C language. *ACM Sigplan Notices* 28, 3 (1993), 201–208. https://doi.org/10.1145/155360.155580

[65] Dennis M Ritchie and K. Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM* 17, 7 (July 1974), 365–375.

[66] Stuart Ritchie. 1997. Systems programming in Java. *IEEE Micro* 17, 3 (May 1997), 30–35. https://doi.org/10.1109/40.591652

[67] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*. 101–112.

[68] Peter H. Salus. 1994. Unix at 25. *BYTE magazine* 19, 10 (Oct. 1994), 75pp.

[69] Sun microsystems and IBM. 1998. JavaOS for Business Device Driver Guide. (June 1998). https://www.oracle.com/technetwork/java/josddk-150086.pdf.

[70] TAPS Working Group. 2019. An Abstract Application Layer Interface to Transport Services. (2019).

[71] TechEmpower. 2019. TechEmpower Framework Benchmarks. (2019). https://github.com/TechEmpower/FrameworkBenchmarks.

[72] Vapor project. 2019. Vapor website. (2019). https://vapor.codes/.