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

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

September 24, 2019

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

C is an awesome language for operating systems!

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

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
1999	19	7		3						1		2
2000	5	3										1
2001	22	6								4		3
2002	15	3		1						1	1	
2003	19	8		2						1	3	4
2004	51	20	5	12							5	12
(...)												
2017	454	147	169	52	26			1		17	89	36
2018	166	81	3	28	8					3	17	3
Total	2155	1184	241	347	124			3		111	350	260
% 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 [1]

[1] 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, but 61% can
- Cutler et al. analyzed 65 CVEs categorized as code execution in the Linux kernel [1]

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.[1]

[1] 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

Can we rewrite drivers in better languages?

- 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?

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

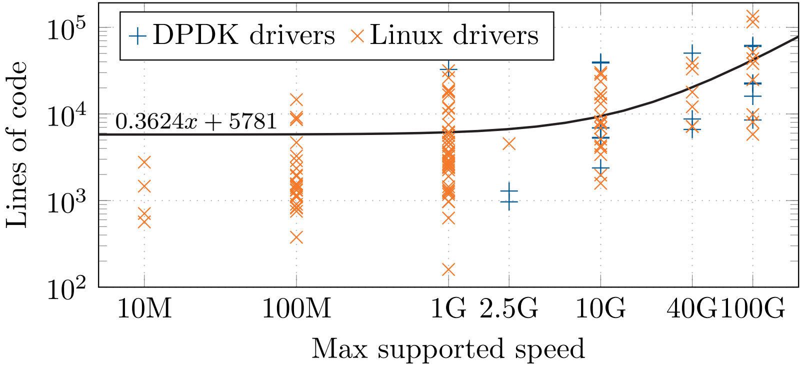
Why look at network drivers?

- 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., QUIC)

Why look at network drivers?

- 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., QUIC)
- Everything mentioned here is applicable to other drivers as well

Network driver complexity is increasing



We wrote full user space network drivers in these languages

C# Swift OCaml Java JavaScript

GO python™

Goals for our implementations

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

Language comparison: Safety properties

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

Table 2: Language-level protections against classes of bugs in our drivers

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

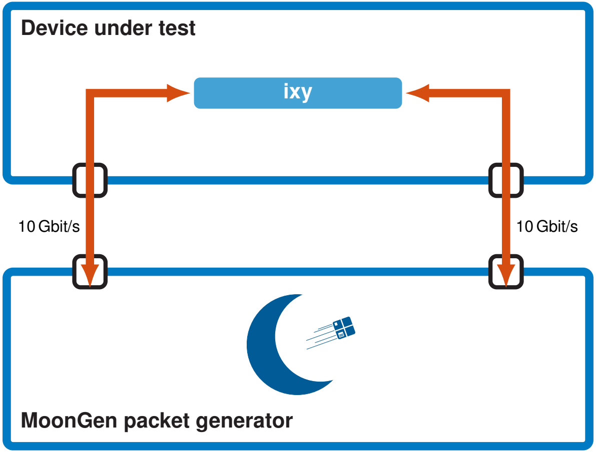
Language comparison: Safety properties

	General memory		Packet buffers		
Language	**Bounds checks**	**Use after free**	**Bounds checks**	**Use after free**	**Int overflows**
C	✗	✗	✗	✗	✗
Rust	✓	✓	(✓)[1]	✓	(✓)[4]
Go	✓	✓	(✓)[1]	(✓)[3]	✗
C#	✓	✓	(✓)[1]	(✓)[3]	(✓)[4]
Java	✓	✓	(✓)[1]	(✓)[3]	✗
OCaml	✓	✓	(✓)[1]	(✓)[3]	✗
Haskell	✓	✓	(✓)[1]	(✓)[3]	(✓)[5]
Swift	✓	✓	✗[2]	(✓)[3]	✓
JavaScript	✓	✓	(✓)[1]	(✓)[3]	(✓)[5]
Python	✓	✓	(✓)[1]	(✓)[3]	(✓)[5]

[1] Bounds enforced by wrapper, constructor in unsafe or C code
[2] Bounds only enforced in debug mode
[3] Buffers are never free'd/gc'd, only returned to a memory pool
[4] Disabled by default
[5] Uses floating point or arbitrary precision integers by default

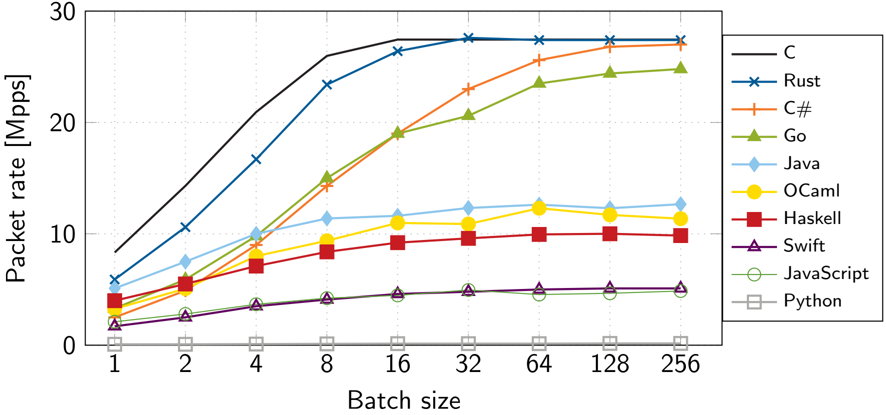
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Performance comparison: Test setup



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

Batching at 3.3 GHz CPU speed (single core)



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

Why is Rust slower than C?

	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

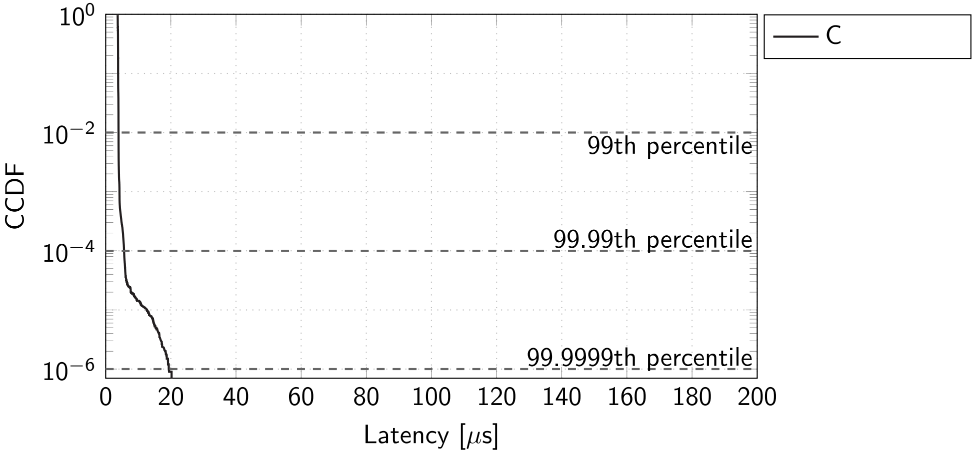
Table 4: Performance counter readings in events per packet when forwarding packets

Why is Rust slower than C?

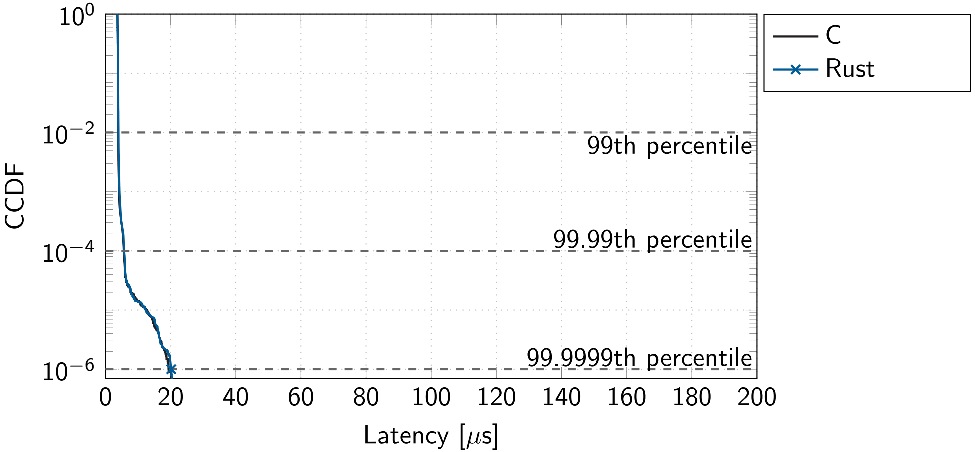
	Batch 32, 1.6 GHz		Batch 8, 1.6 GHz	
Events per packet	**C**	**Rust**	C	Rust
Cycles	94	100	108	120
Instructions	127	209	139	232
Instr. per cycle	1.35	2.09	1.29	1.93
Branches	18	24	19	27
Branch mispredicts	0.05	0.08	0.02	0.06
Store μops	21.8	37.4	24.4	43.0
Load μops	30.1	77.0	33.4	84.2
Load L1 hits	24.3	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 5: Performance counter readings in events per packet when forwarding packets

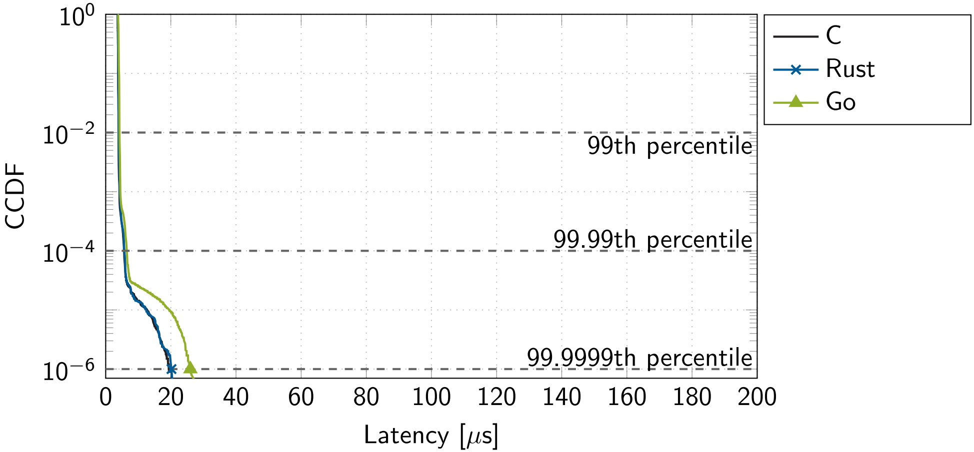
Tail latency at 1 Mpps



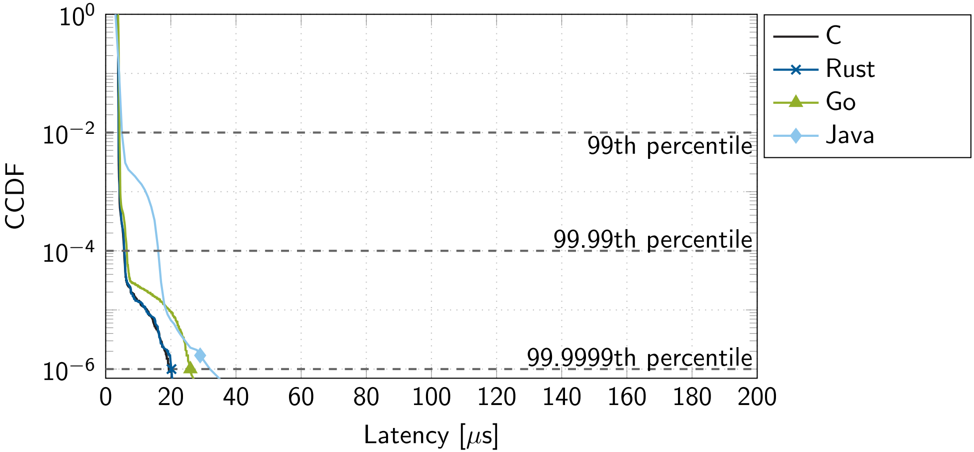
Tail latency at 1 Mpps



Tail latency at 1 Mpps

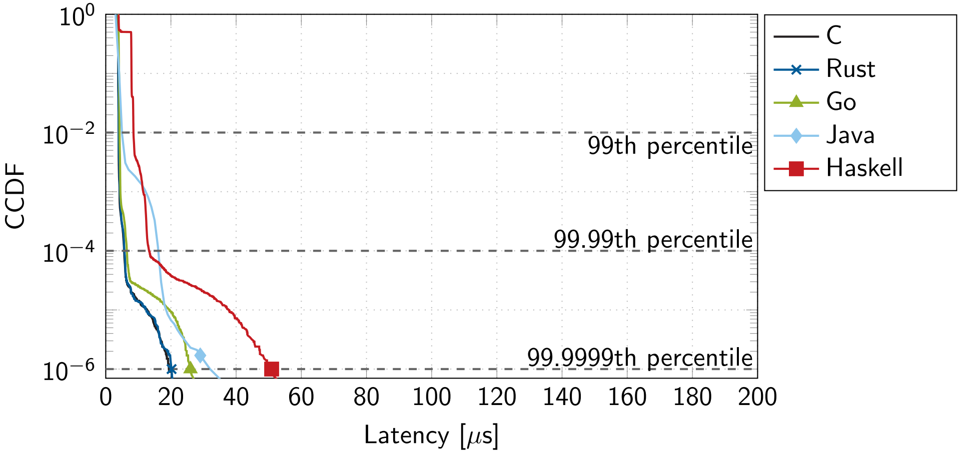


Tail latency at 1 Mpps



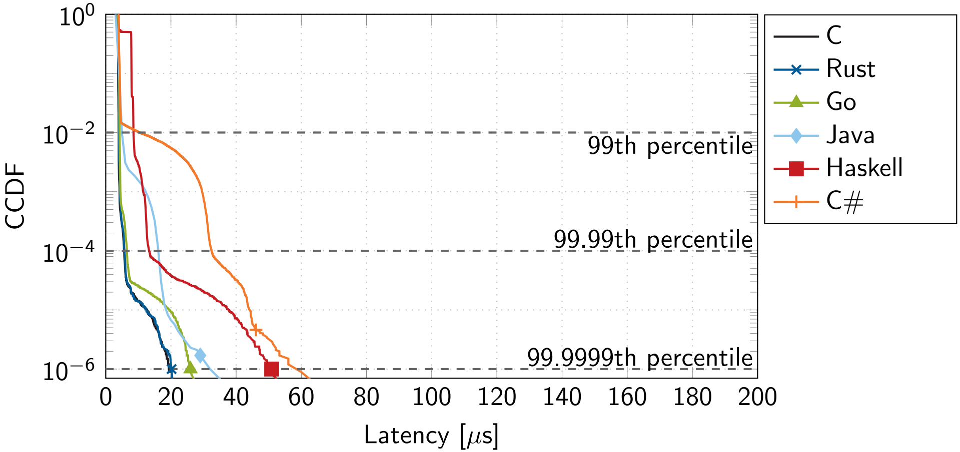
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Tail latency at 1 Mpps

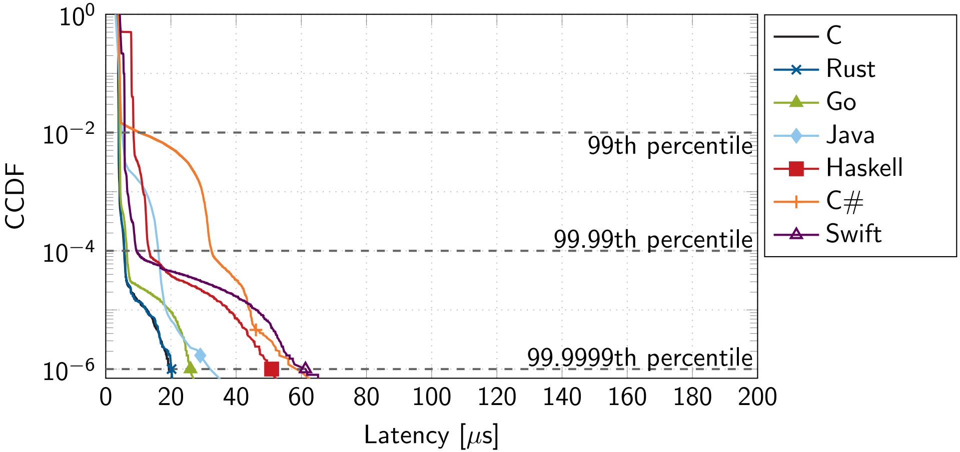


Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

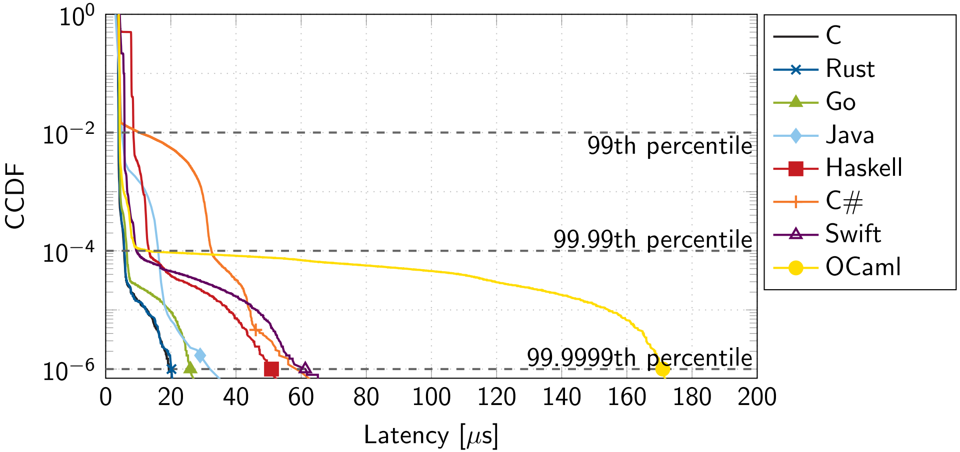
Tail latency at 1 Mpps



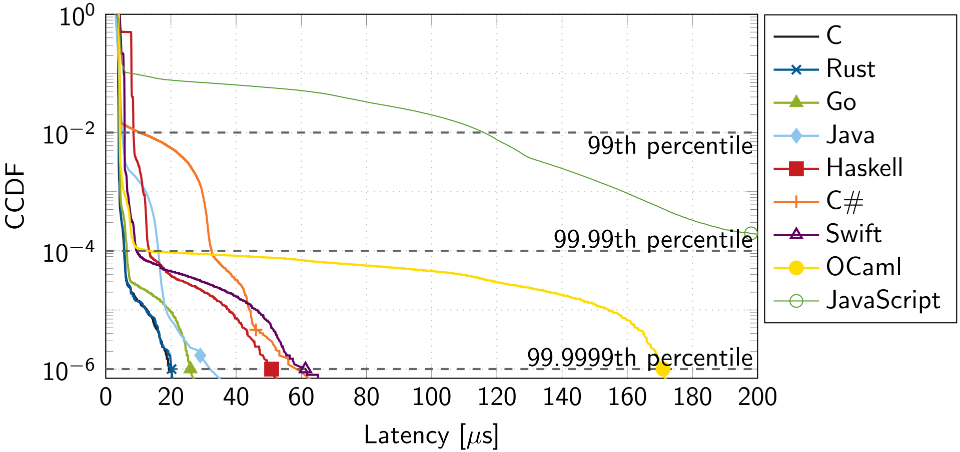
Tail latency at 1 Mpps



Tail latency at 1 Mpps



Tail latency at 1 Mpps



Conclusion: Check out our code

- Meta-repository with links: `https://github.com/ixy-languages/ixy-languages`
- Should your driver really be in the kernel?
- Next time you write a driver: consider a user space driver in a cool language
- Other cool stuff in the paper: details on implementations, latency at higher loads, Java garbage collector comparison, analysis of user space packet processing frameworks used in academia, study of mistakes made in C, and more...

Backup Slides

Languages for code in trustworthy systems

- Rust
  - Fast, no garbage collector
  - Low-level: Easy to reason about performance
  - Safest language of the evaluated languages
- Go
  - Fast, low-latency garbage collector
  - Garbage collector tuned for sub-millisecond latency
  - Easier and faster to write than Rust

Languages for code in trustworthy systems

- Rust
  - Fast, no garbage collector
  - Low-level: Easy to reason about performance
  - Safest language of the evaluated languages
- Go
  - Fast, low-latency garbage collector
  - Garbage collector tuned for sub-millisecond latency
  - Easier and faster to write than Rust
- Other languages
  - Implement critical parts in different languages in redundant systems
  - Functional languages for easier formal verification

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
Java	Object-oriented	Garbage collection	JIT
OCaml	Functional	Garbage collection	Compiled
Haskell	Functional	Garbage collection	(LLVM) Compiled
Swift	Protocol-oriented	Reference counting	(LLVM) Compiled
JavaScript	Imperative	Garbage collection	JIT
Python	Imperative	Garbage collection	Interpreted

Table 6: Language overview

Language comparison: Implementation sizes

Lang.	Lines of code[1]	Lines of C code[1]	Source size (gzip[2])
C	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 7: Size of our implementations (w/o register constants, stripped features not found in all drivers)