KAIST

Advanced Networking Lab

Internet
Network
Architectures

# Memory-safe Network Services Through A Userspace Networking Switch

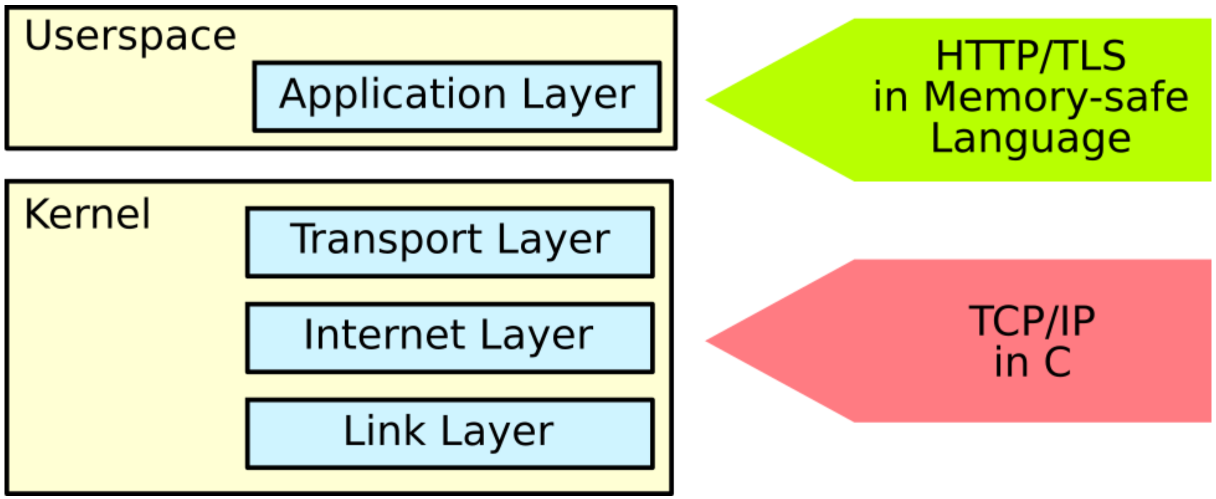
Kai Lüke

Master Thesis

Advisor: Prof. Sue Moon
Co-Advisor: Prof. Anja Feldmann
Dual-Degree Program KAIST/TU Berlin

# Motivation

- Security of network services harmed by *remote code execution* vulnerabilities
- Memory-corruption bugs are the root cause for *remote code execution*
- Applications already use memory-safe programming languages against memory corruption in userspace
- But TCP/IP stack in kernel remains vulnerable
- Idea: Move TCP/IP into memory-safe userspace process



# High-impact Remote Vulnerabilities in Linux

| CVE with CVSS score > 9/10 | Description |
|---|---:|
| CVE-2018-5703 | in-kernel TLS for IPv6 packets has an out-of-bounds write |
| CVE-2017-13715 | flow dissector has uninitialized values exploitable by MPLS packets |
| CVE-2017-18017 | TCP MSS matching in iptables has use-after-free |
| CVE-2016-10229 | UDP packet recv with MSG_PEEK has race |
| CVE-2016-9555 | SCTP out-of-bounds |
| CVE-2016-7117 | recvmmsg use-after-free |

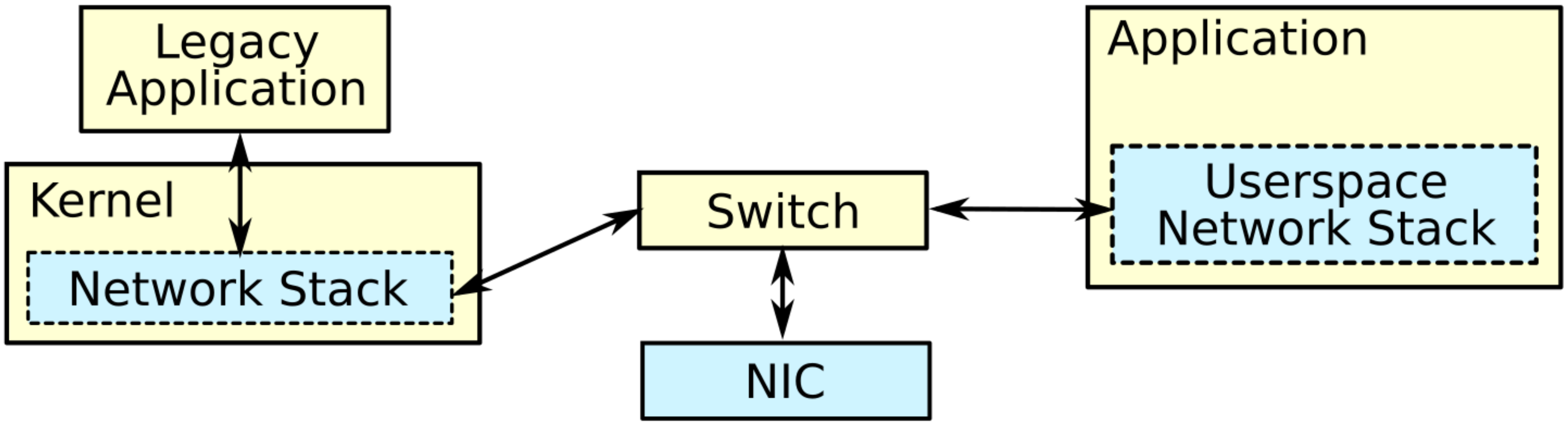
*Not just a problem in Linux:*
**Apple** CVE-2018-4407: ICMP packet may cause out-of-bounds write

# TCP/IP in Userspace

- Implemented as in-app network stack
- Requires access to full Ethernet frames
- Can use kernel bypass and userspace NIC drivers
- Has either exclusive NIC access
  - Dedicated NIC → separate MAC and IP
- Or NIC access shared with kernel/userspace network stacks through switching
  - L2 → separate MAC and IP address
  - L3 → separate IP address
  - L4 → same IP address but separate TCP/UDP ports
    - Appropriate switching layer for OS processes

# Switching Packets for Kernel and Userspace Network Stacks

- Needed because not all applications can be changed to use a userspace network stack
- Keeps kernel network stack active for administrative tasks, updates, and time synchronization
- Allows to first focus on memory safety for publicly available services



# Threat Model

- Assume attacker has knowledge about a vulnerability in our kernel network stack
  - There are ~2 high-impact remote vulnerabilities in the Linux network stack per year
  - Updates may be available or not (third-party, outdated version, custom kernel, zero-day exploit)
- Assume that the attacker uses an Internet endpoint device
  - Can craft malicious IP headers and L4 headers
  - Cannot spoof its source IP address
  - Cannot monitor unrelated packets from the remote host

# Requirements for the Deployment of a Memory-safe TCP/IP Service on Linux

- Allow only memory-safe TCP/IP processing for untrusted packets
  - E.g., L4 switch, firewall, packet monitor
- Allow TCP/IP processing without memory safety only for trusted packets
  - E.g., packets for kernel network stack must be filtered
  - Trust can be defined as policy or heuristic

→ L2 handling and NIC drivers can be trusted code

# Problem Statement for Memory-safe TCP/IP in Userspace

- Provide NIC access to kernel network stack
  - Needs **firewalling of kernel**, e.g., by only trusting response packets for outgoing connections
- Share one IP address with the kernel network stack
  - Needs a **memory-safe L4 switch**, existing L4 switches are not memory-safe
- Provide easy usage for existing applications
  - Needs userspace networking library which
    - **Does not change API** or connection behavior
    - Integrates with kernel **loopback interface**, e.g., for local DB connections

# Existing Solutions for Memory-safe TCP/IP Services

## Memory-safe Operating Systems

| Name | Architecture | Language |
|---|---|---:|
| MirageOS | Unikernel library OS | OCaml |
| HaLVM | Unikernel library OS | Haskell |
| RedoxOS | Microkernel OS | Rust/any |

- Virtual interface connected with L2 bridge
  → different IPs
- Not compatible with Linux but run as VMs on Linux hypervisor
  → unnecessary detour for memory-safe TCP/IP

# Existing Solutions for Memory-safe TCP/IP Services

### Userspace Network Stacks

| Network Stack | Language | Integration with loopback interface |
|---|---|---:|
| mirage-tcpip | OCaml | no |
| HaNS | Haskell | no |
| smoltcp | Rust<br>(single thread with non-blocking IO) | no |

- TAP interface connected with L2 bridge
  → different IPs, need L4 switch for userspace network stack
- Keep benefit of large Linux ecosystem, multiprocess support

# Existing L4 Switches for Userspace Network Stacks

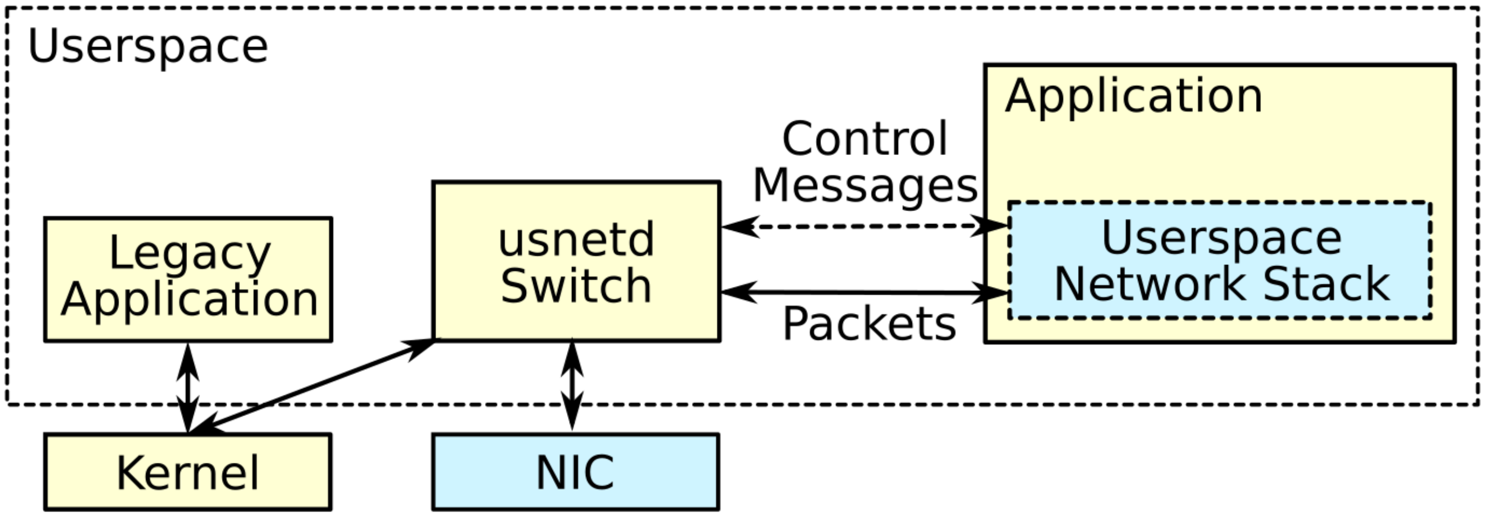
- Let userspace network stacks register TCP/UDP ports to receive
- Forward all other packets to kernel

| **Name** | **Memory-safe** | **Protects Kernel** | **Note** |
|---|---|---|---:|
| MultiStack for VALE | no | no | old netmap version |
| swarm | no | no | requires different kernel IP with VALE |
| TAPM | yes | no | NIC hardware matching through queues |

- Not sufficient for memory-safe TCP/IP services

# Design of *usnetd*: Memory-safe L4 Switch for Userspace Network Stacks

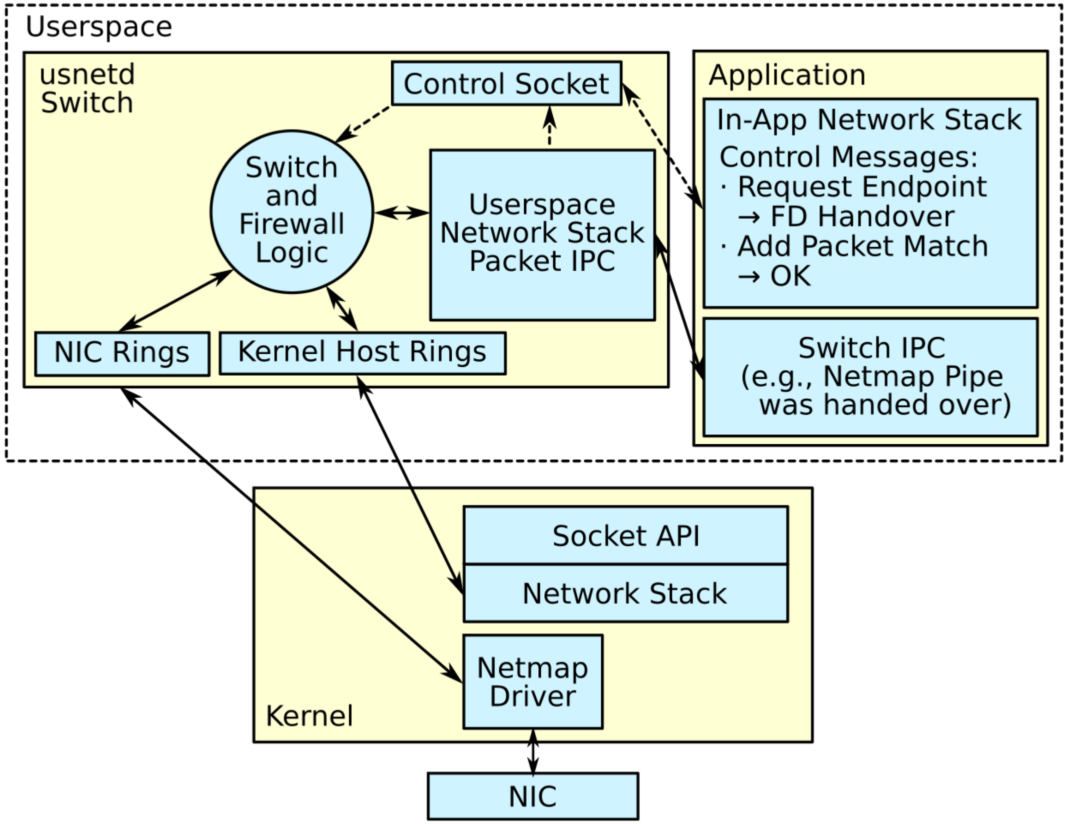
- Shares NIC for userspace network stacks and kernel network stack
- Lets network stacks register which ports to receive
- Firewalls the kernel network stack
  - Allows only outgoing connections, trusts response packets and ARP packets



- Exposes control socket for endpoint setup and port registration
- Automatically forwards response packets without need for port registration
- Provides uniform interface for multiple kernel bypass frameworks as NIC backend
    - DPDK, netmap, macvtap, AF_XDP

# Prototype for *usnetd* on netmap

- Uses netmap pipes as zero-copy IPC channel for packets
- Supports Unix domain sockets as simpler IPC channel
- Uses a single thread for the event loop

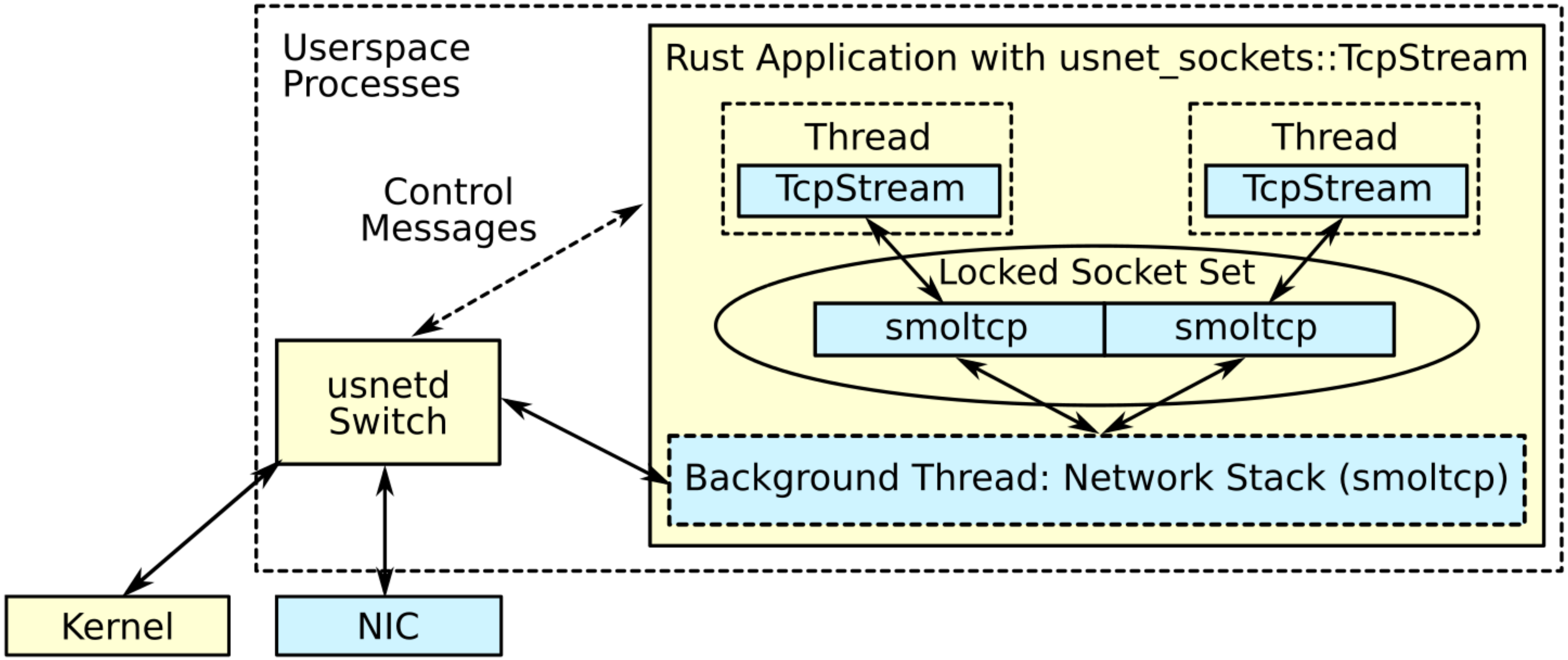


# Design of *usnet_sockets*: Rust userspace networking library

- Uses an in-app network stack
- Provides same socket types as the standard library
- Integrates with the kernel loopback interface
- Requires source code changes for imports/dependencies
- Configurable at runtime
  - NIC access: usnetd, netmap, macvtap
  - IP configuration: passthrough, static, DHCP

# Prototype for *usnet_sockets* on smoltcp

- Provides `TcpStream`, `TcpListener` types
  - Multithread-capable, blocking API
- Uses background thread to wake up blocked socket IO
- Currently relies on one mutex for all smoltcp sockets



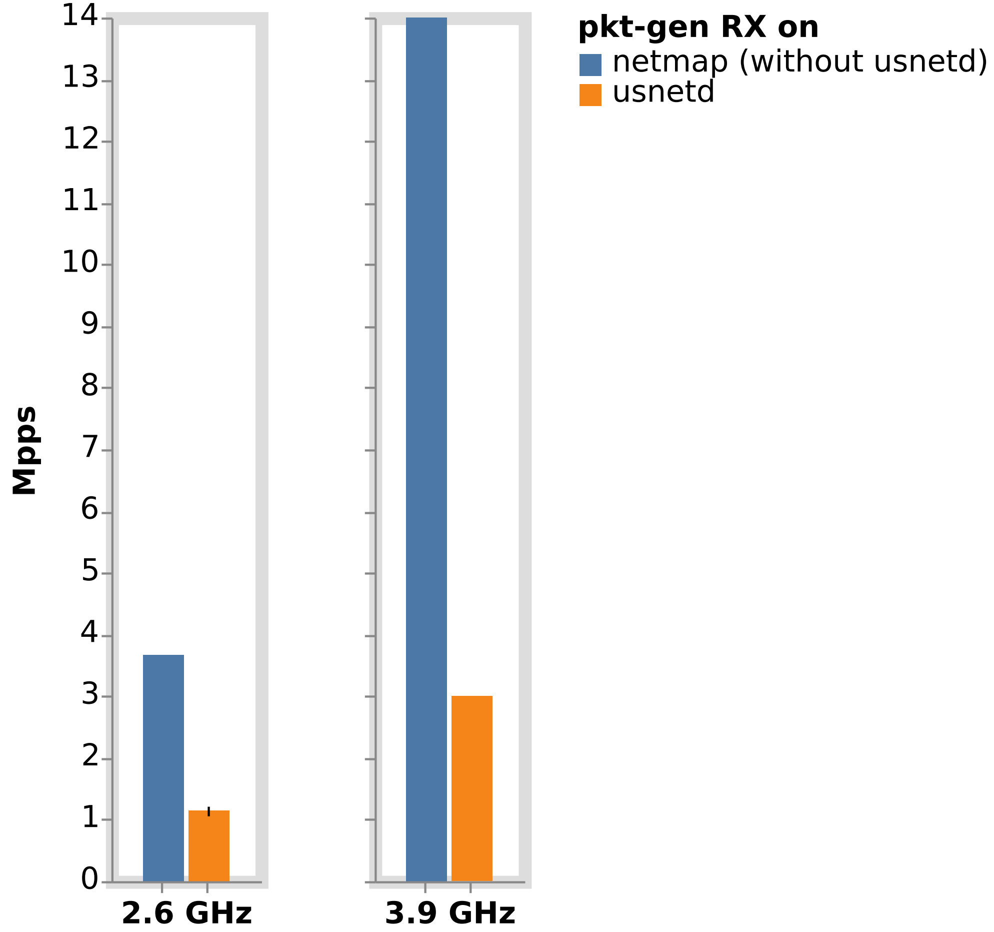
# Performance Evaluation

## Experiment Setup

- Consists of directly connected equal machines
  - Two Intel Xeon X5550 servers at 2.6 GHz
  - Two Intel Core i5-4690 desktops at 3.9 GHz
- Uses Intel 10G NICs with netmap patches for ixgbe driver
- Always uses zero-copy IO

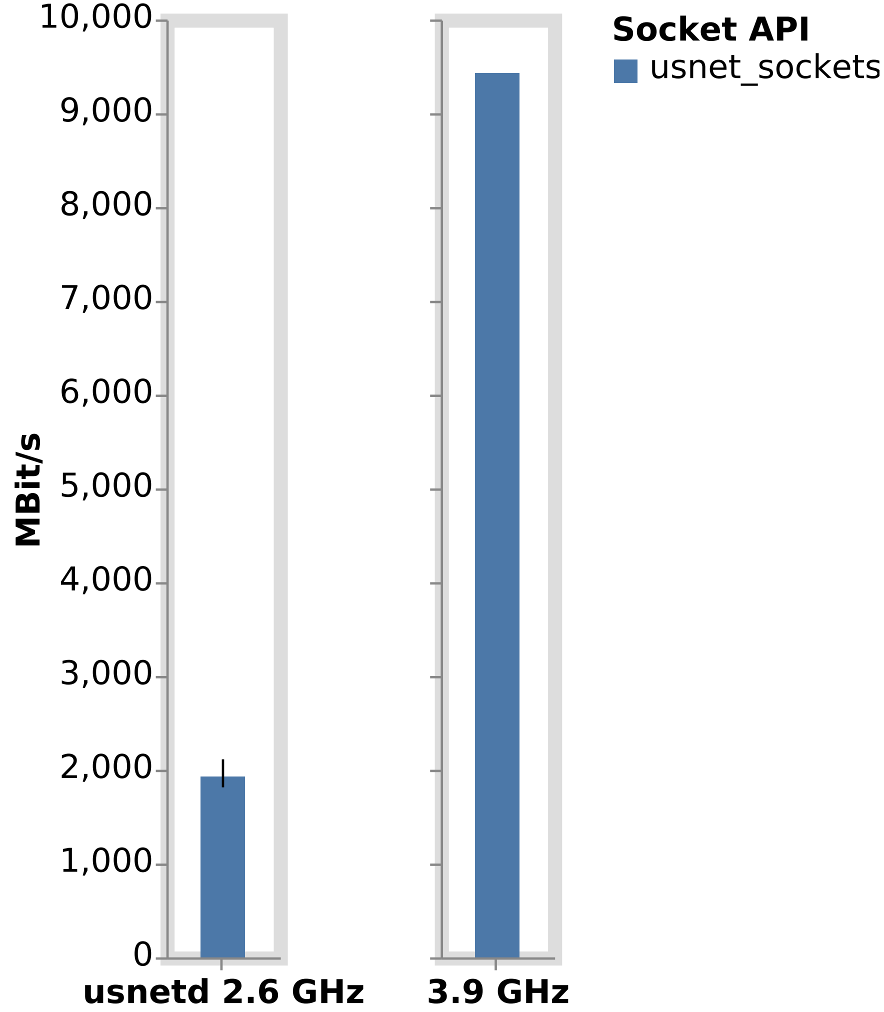
# Packet Matching Speed of *usnetd*

- Measured with netmap pkt-gen connected to usnetd via netmap pipe, sending minimal sized UDP packets
- 2.6 GHz setup cannot achieve line rate for small/middle-sized packets
- 3.9 GHz setup cannot achieve line rate for small packets
- Note: Not linear due to degraded netmap performance on 2.6 GHz



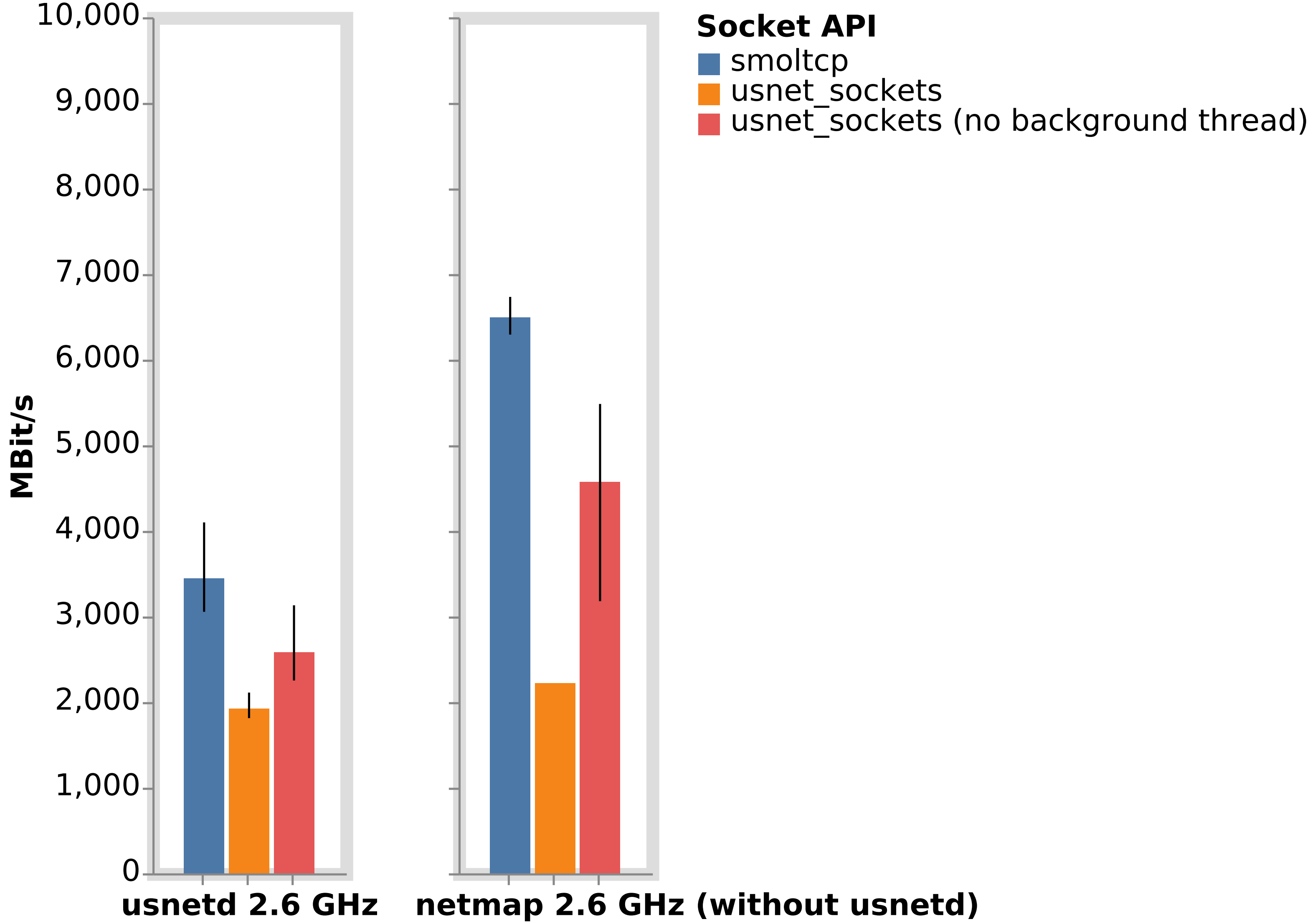
# TCP Goodput of *usnet_sockets* on usnetd

- Measured payload delivery rate, sensitive to packet drops
- Line-rate for large packets with 3.9 GHz setup
- Degraded performance with 2.6 GHz setup
  - Will look in detail what the bottleneck is by measuring overheads
- Linux network stack has always line rate (multicore)



# TCP Goodput: Analysis of Overhead

- Take usnetd out
  - NIC access directly through netmap without usnetd
- Take locking and synchronization in usnet_sockets out
  - No background thread, socket calls do all network IO
- Take blocking API of usnet_sockets out
  - Directly use smoltcp

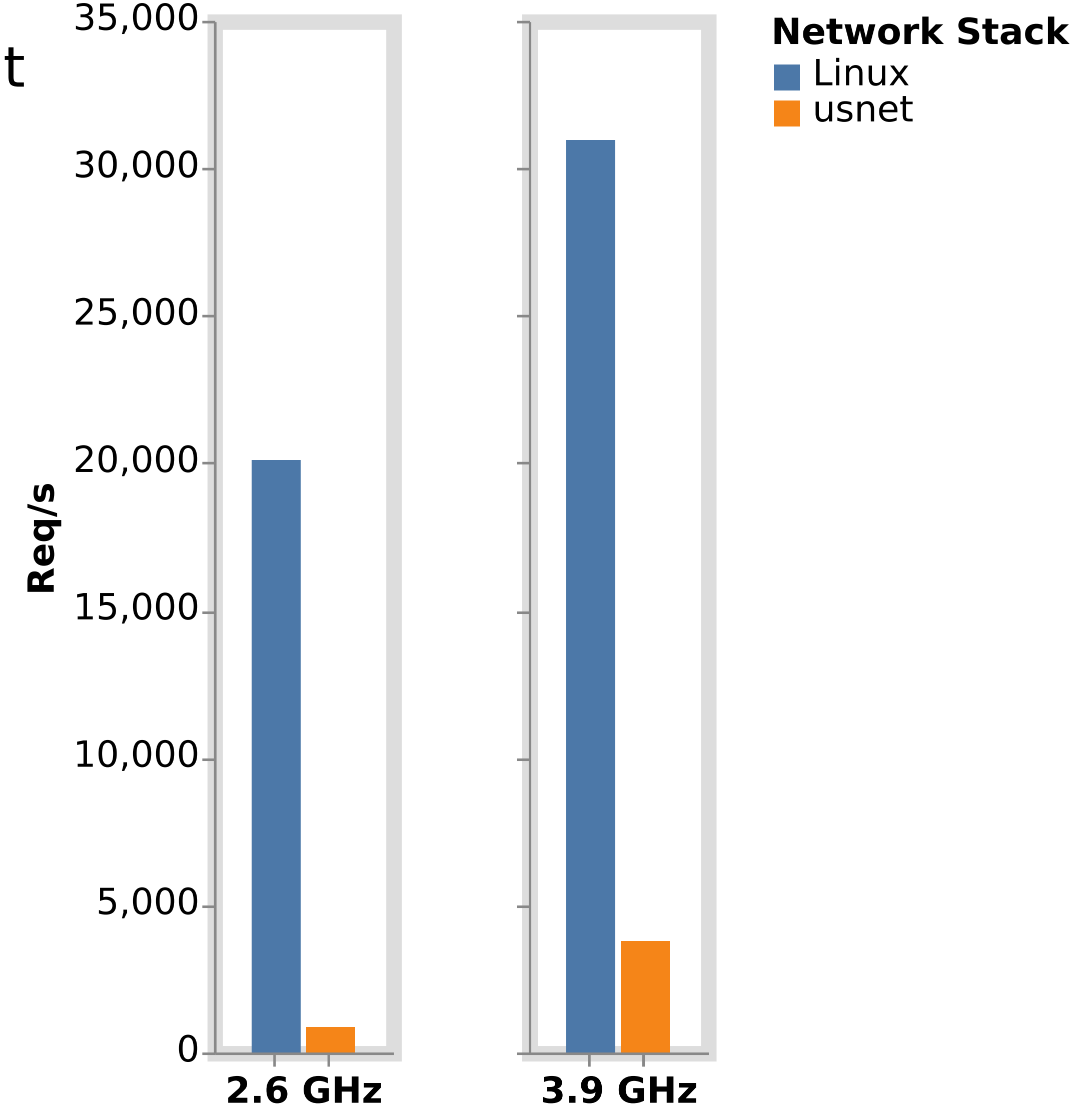

Socket API
smoltcp
usnet_sockets
usnet_sockets (no background thread)
MBit/s
10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
usnetd 2.6 GHz
netmap 2.6 GHz (without usnetd)

- No improvement for usnet_sockets on netmap
    - Multithread synchronization is the main limiting factor
- Not using a background thread is faster
    - Here usnetd limits performance
- Blocking logic has some impact compared to directly using smoltcp
    - But even without usnetd and blocking API goodput is not line rate → Needs smoltcp improvements

# HTTP Request Completion

- Find out limitations of global mutex for multithread scalability
- Measured with Apache Benchmark (ab) as client on Linux network stack
- Client uses 32 parallel short connections, not sensitive to congestion control
- Server spawns thread per connection
- Compared usnet_sockets to Linux network stack

- Scalability impacted by global mutex and inefficient wakeup with lock competition
- Note: Not linear due to degraded netmap performance and more cores on 2.6 GHz



# Required Source Code Changes

- I changed two public Rust libraries to have build flags for using usnet_sockets
- tiny-http is a HTTP implementation on top of the Rust standard library sockets
  - Needed few-lines patch to alter the import statements
- rouille is a web framework using tiny-http
  - Needed build metadata patch to specify build flag for tiny-http
- Port key libraries that use Rust standard library sockets
  - Then no change to applications needed

# Results

- No TCP/IP handling in trusted code base (TCB)
  - TCB consists of NIC driver, netmap, netmap code in Rust, syscalls and data structures of libraries, Rust compiler
- Implementation lacks features in these areas:
  - usnetd: multicast, broadcast, IPv6, IP fragmentation
  - usnet_sockets: DNS, IPv6, timeout, UDP, non-blocking IO, epoll wrapper, routing for multiple NICs
  - smoltcp: congestion control, selective/delayed ACKs, IP fragmentation, MTU discovery, DHCP

# Discussion

- smoltcp: packet loss impacts goodput, needs more TCP features
- usnetd: multi-core scalability needed for line rate
- usnet_sockets:
  - Needs fine-grained locking and notification for multi-thread usage
  - Needs also epoll syscall wrappers for file descriptors to support mio or Tokio
- Only addressed memory corruption, not implementation correctness

# Conclusion

- **usnetd** switch for memory-safe userspace network stacks
  - Shares NIC and IP between kernel and userspace network stacks
  - Protects the kernel network stack
- **usnet_sockets** library for Rust
  - Provides memory-safe TCP/IP without changing application logic
  - Integrates well with the rest of the system through loopback interface
  - Demonstrated ~10 GBit/s TCP goodput
- Together they provide memory-safe TCP/IP for Rust on Linux without changing the application logic
- More optimizations and features needed for parity with Linux network stack

# Future Work

- API completeness
- Macvtap, DPDK, and AF_XDP backends for usnetd
- Programmable switches as alternative to usnetd or as backend?
  - VALE-bpf, AF_XDP, PFQ: need bytecode/script for switching and firewall logic
- Provide libc-compatible LD_PRELOAD wrapper for language agnostic memory-safe network stack