


Technische Universität Berlin, Germany
Faculty IV – Electrical Engineering and
Computer Science, Department of
Telecommunication Systems
Internet Network Architectures (FG INET)

Korea Advanced Institute of Science and
Technology (KAIST), South Korea
School of Computing
Advanced Networking Lab

Master Thesis

Memory-safe Network Services Through A Userspace Networking Switch

Kai Lüke

kailueke@riseup.net

Dual-Degree Master Computer Science

Supervisors: Prof. Anja Feldmann (TU Berlin)

Prof. Sue Moon (KAIST)

January 11, 2019

Erklärung der Urheberschaft

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin 11. Januar 2019

Unterschrift

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin January 11, 2019

Signature

Abstract

A network service needs to be resilient against malicious input from the Internet. Specially programs written in C are prone to memory corruption bugs which are the basis for remote code execution attacks. Memory-safe languages solve this problem for application code running in userspace. The TCP/IP network stack however runs in the operating system kernel, which is written in C and vulnerable to memory corruption. Therefore, this work explored moving the TCP/IP stack into the memory-safe userspace process while providing a compatible API. The process should share an IP address with the kernel and integrate with the kernel's loopback interface. This solution keeps the benefits of a full-featured OS and does not impose different IPs per process or changes in the application logic. I analyzed the requirements for deploying memory-safe TCP/IP services along with the kernel network stack. The existing switching solutions for userspace network stacks do not meet these requirements because they do not handle untrusted packets in a memory-safe language and expose the kernel network stack to untrusted packets. I present a memory-safe L4 software switch that connects multiple userspace network stacks and the host kernel network stack. The switch allows the kernel and userspace network stacks to share an IP address. It also firewalls the host kernel network stack while supporting outgoing connections for updates. To make memory-safe userspace networking easily usable I developed a socket library for Rust. Its in-app TCP/IP stack provides the same socket API types as the standard library and is based on smoltcp. The combination of a memory-safe userspace switch and userspace TCP/IP stack expands the memory-safety of existing Rust web services to the TCP/IP layer with low porting efforts.

초록

사용자 공간의 네트워크 스위치를 사용한 메모리-안전한 네트워크 서비스

네트워크 서비스는 악성 입력에 대해 내구성을 가져야 하는데, 티시피/아이피는 씨로 구현되어 있고 오에스 커널에서 실행되어 메모리 손상 버그에 취약하다. 따라서 본 연구는 호환 가능한 에이피아이를 제공하면서 티시피/아이피 스택을 메모리-안전한 사용자 프로세스로 이동시키는 방법을 소개한다. 기존 방식들은 패킷을 메모리-안전 언어로 처리하지 않아 커널 네트워크 스택이 취약성을 가지므로, 사용자공간 네트워크 스택과 커널 네트워크 스택을 연결하여 아이피 주소를 공유하는 메모리-안전 레이어 4 소프트웨어 스위치를 제시하며 이를 통해 방화벽으로 커널 네트워크 스택을 보호한다. 또한 메모리-안전한 사용자공간 네트워킹을 위해 스몰티시피를 기반으로 러스트용 소켓 라이브러리를 개발했으며 표준 라이브러리와 같은 타입의 소켓 에이피아이를 제공한다. 이러한 메모리-안전 사용자공간 스위치와 사용자공간 티시피/아이피 스택을 결합하여 적은 노력으로 기존 러스트 웹 서비스의 메모리안전성을 티시피/아이피 계층으로 확장시킬 수 있다.

Zusammenfassung

Ein Netzwerkdienst sollte robust sein gegen schädliche Eingabedaten aus dem Internet. Vor allem Programme in C sind anfällig für Speicherkorruptionsfehler welche die Basis für Remotecodeausführungsattacken bilden. Speichersichere Sprachen lösen das Problem für Anwendungscode, der im Userspace läuft. Jedoch läuft der TCP/IP-Netzwerkstack im Betriebssystemkernel, welcher in C geschrieben ist und damit anfällig für Speicherkorruption. Daher untersucht diese Arbeit das Verschieben des TCP/IP-Stacks hinein in speichersichere Userspace-Prozesse bei Beibehaltung einer kompatiblen Programmierschnittstelle. Der Prozess soll sich eine IP-Addresse mit dem Kernel teilen und sich in die Loopback-Schnittstelle integrieren. Diese Lösung behält die Vorteile eines vollausgestatteten Betriebssystems bei und zwingt nicht unterschiedliche IP-Addressen pro Prozess auf oder Änderungen in der Anwendungslogik. Ich analysierte die Anforderungen zum Bereitstellen von speichersicheren TCP/IP-Diensten neben dem Kernel-Netzwerkstack. Die bisherigen Switch-Lösungen für Userspace-Netzwerkstacks erfüllen nicht die Anforderungen, weil sie nicht-vertrauenswürdige Pakete nicht in einer speichersicheren Sprache verarbeiten und auch den Kernel-Netzwerkstack ihnen gegenüber aussetzen. Ich präsentiere einen speichersicheren L4-Software-Switch, der mehrere Userspace-Netzwerkstacks und den Kernel-Netzwerkstack verbindet. Er erlaubt ihnen das Teilen einer IP-Addresse. Außerdem schützt er den Kernel-Netzwerkstack durch eine Firewall, aber erlaubt ausgehende Verbindungen für Updates. Um speichersichere Netzwerkverarbeitung im Userspace einfach nutzbar zu machen habe ich eine Socket-Bibliothek für Rust entwickelt. Ihr TCP/IP-Stack läuft im Anwendungsprozess und stellt die gleiche Socket-Schnittstellentypen wie die Standardbibliothek bereit auf Basis von smoltcp. Die Kombination von speichersicherem Userspace-Switch und Userspace-TCP/IP-Stacks erweitert die Speichersicherheit von bestehenden Webdiensten in Rust auf die TCP/IP-Schicht ohne große Portierungsanstrengungen.

Contents

Glossary

ABI Application Binary Interface

ASLR Address Space Layout Randomization

BPF Berkeley Packet Filter

CVE Common Vulnerabilities and Exposures

DoS Denial of Service

eBPF Extended Berkeley Packet Filter

ICMP Internet Control Message Protocol

IOMMU I/O MMU (Memory Management Unit)

IPC Inter-Process Communication

L2 OSI Model Layer 2 (Ethernet)

L4 OSI Model Layer 4 (TCP, UDP, ⋯)

MPTCP Multipath TCP

MTU Maximum Transmission Unit

NAT Network Address Translation

NFV Network Function Virtualization

TCB Trusted Code Base

TLS Transport Layer Security

POSIX Portable Operating System Interface

QUIC Quick UDP Internet Connections (a TCP+TLS alternative)

1 Introduction

The security of network services is a major issue for private and public infrastructure. Software bugs are one of the many determinants of security. Because developers make mistakes they introduce security vulnerabilities. Given responsible disclosure of bugs developers may be able to fix them in updates before attackers start to use them. 0-day vulnerabilities, however, are only known to an attacker and may remain secret for a long time. Also it is out of reach for a developer whether users apply updates. The availability of updates even depends third-party distributors which, e.g., maintain a customized kernel for their embedded systems. Therefore, it is the developers responsibility to reduce security vulnerabilities. Most at risk are those parts of the system that process untrusted input from the Internet. Fortunately the choice of a memory-safe programming language can already avoid dangerous memory-corruption bugs.

The lack of memory safety for programming languages, such as C, implies that neither compiler nor runtime checks may catch invalid memory access. Instead, the program continues with undefined behavior and can corrupt its memory. Depending on the input it overwrites or reveals code pointers which determine the program flow and other important values. To exploit these memory corruption bugs, attackers craft malicious input. The attacker's goal is often to execute malicious code at the programs privilege level by hijacking the program control flow. The existing countermeasures are unsatisfactory. Mitigations and software testing do not solve the root problem, and software verification cannot be applied easily, as described in Section 2.1.

In a memory-safe language, such as Rust, the compiler guarantees that the code will access only the correct memory areas. Depending on whether memory-safety is enforced at compilation or runtime, the language rejects to compile or at least safely stops code with invalid operations such as out-of-bounds array access, data races, type mismatch, or lifetime mismatch of references. This solves the root cause for code execution attacks or memory leaks which grant access to sensitive data. It is, therefore, beneficial to write a network service in such a language. In the userspace code no memory corruption is possible, from parsing and processing HTTP requests down to TLS. Instead of trusting the developer's code the only trusted code base (TCB) is the compiler and the standard library.

The standard libraries of memory-safe languages, however, are wrapping the system calls on common operating systems like Linux. This implies not only trusting the standard library but also the

kernel since any vulnerabilities in kernel space impact the whole system. There is no vertical support for memory-safety in all protocol layers. For a network service the kernel's TCP/IP implementation is in a critical position because attackers can directly feed it with malicious input over the Internet. While this also holds for client systems with malicious server responses or targeted attacks, they are less exposed than public servers. Moving the kernel's TCP/IP implementation out of the TCB prevents against future memory-corruption security vulnerabilities like those discovered in the last years (cf. Section 2.2).

A memory-safe network service as regular userspace process just needs direct network access to replace the critical path of TCP/IP handling in the kernel with an own userspace network stack. The API of the userspace network stack should not require changes in the application logic and integrate well with the kernel's loopback interface for local connections. Bypassing the kernel for direct NIC access is already established for high-performance network services but not yet for memory-safe TCP/IP in userspace. Because system updates, administrative tools, and time synchronization rely on the kernel network stack, the kernel should not lose network access. A switch as shown in Figure 1.1 is needed to share the NIC between kernel and userspace network stacks. L2 switching imposes different IPs for the kernel network stack and each userspace network stack. Special L4 switches for userspace network stacks avoid this and allow to share one IP but currently have no memory-safe packet matching. In addition, the kernel network stack needs protection from untrusted packets because its lack of memory safety undermines the gains of memory safety in userspace networking. Based on a threat model I derived the requirements to deploy memory-safe network services while sharing one IP with the kernel network stack (cf. Section 3.1).

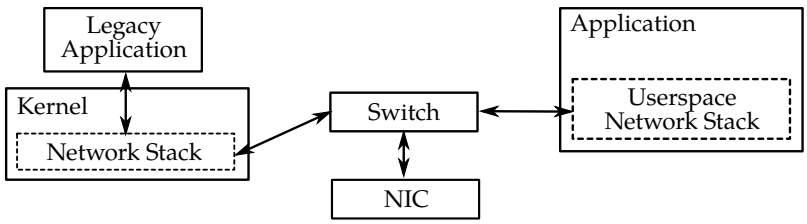

Figure 1.1: A software switch shares the NIC between kernel and userspace network stacks

This work explores memory-safe userspace networking and its challenges, analyzes prior work, designs a solution and evaluates a prototype implementation. I address the following problems at once: How to deploy multiple userspace networking services together on the same IP, how to protect the kernel network stack from untrusted packets, and how to provide an easy path to memory-safe TCP/IP handling for an existing Rust code base by using the same API and connection behavior as the standard library. It relates to existing solutions as follows.

Some clean state approaches for memory-safe operating systems such as unikernel cloud VMs exist. The solutions presented in Section 2.4 usually run as VM on a Linux hypervisor. Section 3.3 analyzes the shortcomings of the current solutions and concludes that unikernels are a detour which is

not necessary for memory-safe networking. Adding memory-safe userspace networking to regular processes is a less invasive change than the unikernel approach and keeps the benefits of common operating systems such as Linux.

There are many ways of achieving packet access for userspace networking and to some extend they overlap with how guest VMs do packet access. Chapter 2 introduces kernel bypass frameworks, software switches and others. For a userspace process, sharing the same IP with other processes should be preserved regardless of the fact whether they use an own network stack or the host kernel's network stack. Prior art does not fill this gap for memory-safe networking, as analyzed in Section 3.4.

To address these considerations, I designed a memory-safe software switch for userspace network stacks (cf. Section 4.1). As protection for the host kernel network stack I decided for a firewall which only matches response packets for outgoing connections. Section 5.1 presents a prototype, its implementation choices, and the current limitations. The evaluation in Section 6.1 compares the performance and states room for optimizations.

For some memory-safe language there are HTTP servers, web frameworks, and TCP/IP implementations. However, there is currently no general way to use them together on a common Linux OS. They do not provide a seamless conversion from using the kernel's network stack towards memory-safe TCP/IP.

To make memory-safe userspace networking more accessible I designed a Rust socket library for memory-safe userspace network services in Section 4.2. It provides the same socket API types as the Rust standard library based on the TCP/IP stack smoltcp. The socket API integrates well with the kernel loopback interface. The library shows the advantages of the switch because the process keeps the same IP as the kernel network stack and can connect with other network stacks on the system. Section 5.2 describes the prototype and its limitations. The evaluation in Section 6.2 and 6.3 presents performance measurements and an example port of a HTTP library.

The discussion in Chapter 7 gives a critique on the proposed design and its limitations and argues for different solutions and future work. The switch combined with the userspace network stack allows existing Rust web services to benefit from a memory-safe TCP/IP layer with minimal porting efforts. The solution interacts with the kernel network stack and other userspace networking services in the same way as if there was no userspace networking.

2 Background

Sections 2.1, 2.2, and 2.3 describe why the security problems of memory corruption the network stack have to be addressed with a memory-safe language. Sections 2.4, 2.5, 2.6, and 2.7 introduce existing solutions for memory-safe unikernel VMs, memory-safe operating systems, and network stack implementations in memory-safe languages. Section 2.8 presents operating system designs that facilitate userspace networking. Sections 2.9, 2.10, 2.11, and 2.12 cover packet access and switching solutions in Linux, kernel bypass frameworks, software switches for Linux, and hardware switching in the NIC. Userspace networking can use these approaches for exclusive or shared packet access. Section 2.13 presents userspace network stacks and which kind of APIs they provide to the application.

2.1 Mitigations, Software Testing, and Software Verification

The programming languages C or C++ are not memory safe. To address this, several mitigation techniques have been developed to lower the chances for successful exploits. Mitigations often focus on a single symptom of memory corruption under certain assumptions. In practice attackers find ways to circumvent the mitigations. ASLR (Address Space Layout Randomization), for example, makes it harder for the attacker to know the memory locations. It relies on information hiding and assumes that code pointers remain secret. Attackers can defeat ASLR through information leak bugs (invalid memory reads), probabilistic methods, or side-channel attacks. Side-channels are hard to avoid and can even the recently discovered Spectre flaws in CPU hardware have to be considered. Remotely breaking ASLR and reading memory with Spectre variants was demonstrated by observing the varying latency [1]. Shadow Stacks mitigate hijacking the return addresses. CFI (Control-Flow Integrity) mitigates hijacking code pointers for function calls. Code pointer authentification mitigates hijacking code pointers in general. DFI (Data-Flow Integrity) also mitigates attacks on non-control-flow data. Strong mitigations are often only partly applied or unused in practice.

Software testing and fuzzing can discover many bugs when they are integrated into the development process. But they only reduce the number of bugs and cannot prove the absence of bugs.

In contrast, formal software verification methods are able to proof correct memory usage. However, they are often not integrated into the development process and, therefore, often remain unused. This

is specially a problem for manual proofs involving theorem provers. Automatized tools for static analysis may be used similarly to testing and fuzzing. There may be a high initial cost if the analysis needs modeling of certain system aspects. A model also requires maintenance afterwards. Bug finding tools which do not require any modeling are easier to use. But since they lack knowledge about code interactions they either cannot detect some problems or generate many warnings which are false positives. False positives are a general problem if an analysis should detect all problems and have a short runtime. The developer needs to classify the warnings and verify manually if there is a problem. This may lead to ignoring the warnings [2]. It is easier to focus on concrete security problems when the analysis generates test cases which demonstrate failure. Klee [3] and angr [4] are examples for this. However, analyzers focus more on support for userspace applications than operating system kernels. Also the resource requirements are too big for a tight integration into the development flow.

As result even well-established and mature software, such as the Linux kernel, may still contain memory corruption bugs which lead to security vulnerabilities.

2.2 Security of the Kernel Network Stack

The attack surface of the kernel's network stack consists of parsers, protocol state logic, firewall actions, and management of incoming data in socket buffers. The possible impact of memory-corruption bugs is often not investigated when they are fixed because demonstrating remote code execution is hard. Among those bugs identified as security vulnerabilities in the CVE (Common Vulnerabilities and Exposures) database for Linux [5] many memory-corruption bug descriptions only name system crashes and "unspecified other impact", thus, leaving open if code execution is possible or not. To find possible remote code execution bugs in the CVE database one, therefore, has to look for buffer overflows, out-of-bound access, NULL/invalid-pointer dereference, stack consumption, denial-of-service (due to invalid memory access or corrupted memory), uninitialized values, integer overflows for array indices, use-after-free, double-free, data race bugs or those simply called memory-corruption. Bugs as in Table 2.1 underline that many Linux systems without updates are at risk even without considering secret zero-day exploits.

Other kernels besides Linux also have a history of memory corruption bugs in the network stack. The IPv6 fragmentation vulnerability CVE-2007-1365 in the security-oriented OpenBSD kernel is a good example for the severity of memory corruption, in particular, because the developers first disputed that remote code execution is possible until the finders provided a proof of concept exploit [6]. Looking at client systems, the CVE-2018-4407 for the XNU kernel causes a crash or remote code execution affecting many Apple consumer devices due to a buffer overflow when generating ICMP responses [7].

Table 2.1: Memory corruption bugs in the Linux network stack 2014-2018, CVSS score > 9/10

CVE Number	Description
CVE-2014-0100	IPv4 fragmentation data race with use-after-free causes crash or "unspecified other impact"
CVE-2014-2523	DCCP packet header pointer bug causes crash or code execution
CVE-2015-1421	SCTP use-after-free slab memory corruption causes crash or "unspecified other impact"
CVE-2015-8787	netfilter NAT NULL-pointer deref causes crash or "unspecified other impact"
CVE-2016-7117	recvmmsg syscall use-after-free code execution
CVE-2016-9555	SCTP out-of-bounds access causes crash or "unspecified other impact"
CVE-2016-10229	UDP packet recv syscalls with MSG_PEEK trigger an unsafe checksum calculation causing code execution
CVE-2017-18017	TCP MSS matching in iptables use-after-free memory corruption causes crash or "unspecified other impact"
CVE-2017-13715	Flow dissector has uninitialized values exploitable by MPLS packets causing crash or possibly code execution
CVE-2018-5703	in-kernel TLS for IPv6 packets has an out-of-bounds write causing crash or "unspecified other impact"

The Linux networking code for TCP/IP is not frozen but continuously evolving which bears the risk for new bugs. Also the possible inclusion of TCP options such as MPTCP (Multipath TCP) or tcpcrypt can introduce bugs in new or modified code. Merging such changes while avoiding to rewrite much existing code is a slow process [8]. In contrast, development of userspace protocols on top of UDP such as QUIC does not have these hurdles –in this regard having protocol implementations in the kernel hinders innovation.

A recent trend in the Linux kernel acknowledges the security problems of memory corruption by partly moving the networking code to isolated eBPF bytecode. The eBPF bytecode VM of the kernel ensures that the program terminates and correctly accesses the memory. Examples are the flow dissector [9] which extracts information of all packets, and the bpfilter firewall [10]. It is, however, unrealistic to move the complex handling of TCP/IP to eBPF bytecode due to the limitations of the architecture which is mainly meant for quick packet parsing decisions.

In 1999 a TCP implementation in Prolac was used as an alternative TCP layer in the Linux kernel for research purposes [11]. Prolac is a typed language for protocol implementation inspired by special protocol languages such as Esterel, LOTOS, and RTAG. It compiles down to C and supports interfacing with C through embedded code statements. Disadvantages of this approach, however, are the complexity due to code generation and limitations, as only TCP is safe but neither IP handling nor the socket syscalls. Changes in the kernel can also invalidate assumptions in the unsafe glue code. A rewrite of the full network stack in a memory-safe language is not only a major undertaking but also has likely no future for inclusion because the only development language for the Linux kernel is C.

Fuzz testing for the Linux kernel is mainly done by generating random syscall parameters with trinity [12] and syzkaller [13]. Syzkaller also supports fuzzing with mutated network packets.

Several approaches for formal verification in the Linux kernel exist. Sparse is a static code analysis tool for Linux [14]. The Linux code has annotations which sparse uses to check for correct pointer types and locking calls. Smatch is built on top of sparse as a framework for, e.g., value analysis [15].

The stanse analyzer uses abstract interpretation. Symbiotic is an improvement on stanse which additionally uses symbolic execution with Klee to reduce the false positives [16]. It was used to check for errors in correct paring of locks and allocation calls, and correct pointer usage in the Linux kernel. The Linux Verification Center uses the frama-c static analyzer for astraver to check against integer overflows and other properties [17]. Model checking is used to verify kernel modules for Linux driver verification [18]. While their tool finds leaks, race conditions, uninitialized variables, and null pointer dereference, it could not find all bugs present in the code. CMC is a model checker that was used to test the TCP implementation behavior but not memory issues [19]. The Coccinelle [20] analysis tool is established in the kernel development process and allows to rewrite code by semantic patches and, therefore, discover bugs where changes are needed. Some semantic patches are applied on the Linux kernel code on a regular basis. Also Coverity [21] is used regularly to find potential bugs in the kernel [22]. Similarly to fuzzing, all these static analyzers reduce the number of bugs but are currently not ensuring memory safety. This may either be due to their limitations or due to the fact that they are not used often enough and all their reported bugs fixed.

2.3 Fault Isolation

Besides preventing memory-corruption faults, one can also accept their existence and isolate the impact. Microkernel operating systems are a good candidate for this approach because the network stack can run as restartable service in userspace with restrictable permissions. The NewtOS modification of Minix goes even further by splitting the network stack into multiple processes without major performance impact [23]. Instead of using processes for isolation, the network stack can run in another VM, called by the principle *network stack as a service* [24]. Yet both approaches do not address the root cause for unsafety. A successful attack can either disrupt the service with many crashes (DoS attack) or gain code execution with privileges according to the isolation granularity. From this level the attack surface increases with the possibility of local privilege escalation attacks and examining other services in the internal network. The attacker can also act as man-in-the-middle by intercepting or modifying traffic since the network stack is now under her control.

2.4 Unikernels and Memory Safety

Memory-safe network stacks are often part of memory-safe operating systems as discussed in the later sections. These operating systems usually rely on just a small trusted code base, e.g., in assembly or C, for initialization and language runtime code. This allows for the network stack, filesystem, and device drivers, etc. to be written in a memory-safe language.

A major obstacle with this approach is the lack of POSIX compatibility, features and drivers. At least the set of needed drivers is reduced by relying on VM hypervisors. But it comes at the cost of a larger trusted code base, i.e., the Linux kernel and its virtual IO interfaces.

It is a huge and complex task to write a general-purpose multi-user operating system in a memory-safe language. Single-purpose cloud VMs, however, do not require a full multi-user OS and often not even multiprocess support. They are candidates for a unikernel architecture [25]. A unikernel is an application compiled as a kernel binary by replacing the usual syscalls to a separate kernel with calls into a library OS.

MirageOS [26] is a memory-safe unikernel framework with TCP/IP and drivers implemented in OCaml. It spans the use cases of cloud web services, firewall VMs, etc. The Jitsu toolstack for the Xen hypervisor [27], for example, starts MirageOS unikernels on-demand according to incoming DNS requests. It also performs the initial TCP connection setup and then hands the connection over when the VM is ready. Each unikernel needs its own subdomain and IP. Internal communication, however, is possible via shared memory channels. Another memory-safe unikernel framework is HalVM for Haskell which also includes TCP/IP in Haskell [28], e.g., used for honeypot deployments.

Most other unikernels do not include a memory-safe network stack. They are written in C/C++ target traditional Linux applications with a POSIX libc. The OS^V unikernel implements its own network stack in C and has full multithread support. IncludeOS implements its own network stack in C++ as well. Hyperkernel is a verified kernel but uses lwIP in C as userspace system service [29]. The HermitCore port to Rust also uses lwIP [30]. The Rump kernel concept enables code sharing between kernel and userspace implementations through a hypercall interface [31]. NetBSD adapted this approach and, thus, its drivers and network stack may be compiled as userspace libraries. Microkernel operating systems such as HURD and GenodeOS can, therefore, include the NetBSD drivers. A special use case is the rumprun unikernel on bare metal or as seL4 guest [32, 33]. EbbRT is a framework to offload application instances to unikernel VMs with an own network stack in C [34]. The unikernel instances stay connected with the application process to relay some functionality.

2.5 Prior Memory-safe Networking

Historically remarkable for hardware-assisted runtime type checking were the LISP machines from MIT AI Lab, LMI, and Symbolics. The whole operating system was written in LISP, including the drivers and network protocols [35]. At first LISP machines only supported Chaosnet LAN but the later Genera machine had one of the earliest Ethernet and TCP/IP implementations [36, 37].

In contrast to LISP's dynamic typing or weakly-typed languages such as C, languages with a strong static type system allowed for safety guarantees at compile time. An example for a safe systems programming language is the ALGOL-like Ada from 1980. A similar case is Modula-3, that also

allowed mixing of compiler checked safe code and unsafe code for necessary low-level operations. The SPIN microkernel project in Modula-3 had a memory-safe user-extensible networking stack [38, 39].

The functional language ML also guarantees the absence of type and memory errors at compile time. It was not designed for systems programming but the Fox project did pioneering work since 1991 to write an operating system in the slightly extended SML dialect. The network stack FoxNet [40, 41, 42] was tested with raw sockets in userspace on Linux and on the Mach 3 microkernel. The Hello project ported it on bare metal by leveraging Linux boot code to start the SML runtime, and a NIC driver in SML [43]. The ML/OS project also ported the SML runtime to the Flux OSKit but relied on the included C NIC driver [44].

Similar research was done for the type-safe functional language Haskell. The Haskell operating system hOp/House included drivers and a network stack [45].

The Singularity research project used another approach for a new operating system design based on software-guaranteed isolation. It included memory safety for networking in a C# dialect [46]. Midori and Verve were similar kernel research projects based on type and memory safety. Verve is verified through use of a typed assembly language and Hoare logic proofs [47].

The domain specific languages for protocols SPL [48] and MPL [49] from the Melange project have TCP/IP implementations. They compiled to OCaml which is a dialect of the functional language ML.

Erlang is a functional runtime type-checked language and has TCP/IP implementations dating back to 1996 and 2005 [50]. The LING Erlang runtime on the Xen hypervisor, however, uses the C TCP/IP implementation lwIP [51].

Memory-safe network stacks for embedded systems, i.e., single thread, exist in Java [52] and the functional LISP-like language Scheme [53].

The PFQ kernel bypass framework provides a userspace library for networking in Haskell.

2.6 Networking with Go and Lua

Existing TCP/IP implementations in Go are the userspace network stack GoNet [54], the network stack of the monolithic POSIX-like Go kernel Biscuit [55], and the network stack of Google's experimental Fuchsia OS [56]. Fuchsia's network stack is most complete and also present in gVisor. The the container runtime gVisor implements the Linux syscall ABI via Go in userspace [57]. Containers with gVisor are especially interesting because they do not require recompilation. Since the main motivation is containment of untrusted local user code and not untrusted TCP/IP packets from the

outside, gVisor just uses RAW sockets on veth interfaces which means that the Linux kernel is still responsible to handle and dispatch the incoming traffic. The biggest limitation is that Go programs are only memory-safe if there are no data races. A compiler flag for runtime detection and safe handling of race error exists but "memory usage may increase by 5-10× and execution time by 2-20×" [58].

Snabb [59] is a Lua framework for NFV applications, such as VPN implementations, but also software switches for VMs. Currently it only has a work-in-progress TCP implementation ported from smoltcp. The drivers in Snabb are written in Lua as well, a dynamically-typed language with memory-safety guaranteed for single-thread usage.

2.7 Memory-safe Networking in Rust

Rust as a high-level language for low-level programming. It aims to be a memory-safe alternative to C/C++ by not requiring a language runtime and garbage collector. The core language constructs and types have compile-time guarantees for the absence or memory corruption and data races. Custom unsafe code lines are possible but leave the memory correctness guarantees of the compiler. This is often necessary for certain data structures in the standard library as well as interfacing with the OS kernel or C libraries. These parts form the trusted code base in addition to the used OS kernel interfaces. It is an ongoing effort to verify the usage of unsafe Rust code blocks [60]. Rust is significantly faster than traditional type-safe languages like OCaml and Haskell in the Computer Language Benchmarks Game with a performance almost similar to C/C++ [61]. While Rust was initially used to write the Servo web browser engine, it also gained the interest of system programmers and researchers that focus on properties beyond performance and memory safety [62].

The most complete network stack in Rust is smoltcp [63], initially written for embedded devices but also working with Linux on TAP devices and RAW sockets. It replaced the simple TCP/IP implementation of the Rust microkernel operating system RedoxOS [64]. An optional firewall component for the CAmkES embedded operating system on the microkernel seL4 uses smoltcp for packet parsing and validation [65]. The seL4 kernel also supports network services in Rust with smoltcp.

Other incomplete TCP/IP implementations are the network code of the discontinued Rust unikernel QuiltOS [66], the work-in-progress usrnet [67] and the recovery network stack in Fuchsia OS [68]. TockOS is a platform for embedded Rust development but currently does not include a full TCP/IP network stack [69, 70, 71]. The eDSL Rust compiler plugin is a domain specific language for annotation of packet types [72]. This work also demonstrated kernel bypass with netmap [73]. The netmap syscall interface provides direct access to NIC packets and also allows forwarding the kernel network stack. The low-level networking library libpnet provides many packet types and works on RAW sockets and netmap [74].

NetBricks [75] is a framework for memory-safe userspace network functions in Rust. It uses the kernel bypass framework DPDK [76] on Linux for direct NIC access. DPDK implements drivers in userspace and takes the NIC away from the kernel. Another example for memory-safe userspace networking is a Rust NAT service (Network Address Translation) on top of DPDK [77]. Linking to the C library DPDK for userspace drivers can be avoided as the Rust port of the ixy ixgbe driver shows [78].

2.8 Alternative OS Designs for Userspace Networking

Major research of userspace networking, more-suitable OS architectures, and programmable NICs started in the 90s. Motivations were latency reduction in compute clusters, performance benefits of application-specific networking, e.g., for copy reduction or closer protocol integration, and composable modular network stacks.

Specially early microkernels had significant IPC overhead for regular read/write calls. Replacing the networking system service of Mach 3 by in-app networking brought performance similar to monolithic kernels. Two early works [79, 80] left connection management in a registration service but introduced shared packet buffers between application and the network driver for the packet transfer. Thus, the demultiplexing of incoming packets took place in the kernel but the TCP/IP stack moved to the application.

Another early work introduced kernel bypass for a UNIX kernel on a custom ATM-like LAN with channel identifiers [81, 82]. The modified driver demultiplexed packets to multiple userspace network stacks as well as the original kernel network stack. A scatter-gather approach and batched *syscall scripts* minimized the overhead of working with the exposed memory pools. The TCP/IP stack ran in a separate process from the application in order to handle the event timers and support BSD socket semantics.

The U-Net architecture motivated userspace networking for cluster computing with Active Messages on an ATM LAN [83, 84, 85]. It virtualized the NIC as endpoint for each application. This endpoint can either be emulated by the kernel or with hardware support directly mapped mapped from the NIC. Either kernel or NIC do the packet demultiplexing. Applications fill the virtual NIC's control queues with packet data or memory pointers for DMA. Both polling or event notification are available. For Ethernet, however, an additional filter would be needed to determine the receiving application for a packet by, e.g., matching TCP/UDP ports or MAC addresses.

U-Net/SLE introduced Safe Language Extensions for the NIC. Applications could upload Java code for packet processing such as flow control, acknowledgements, and retransmission [86].

Application-specific Safe Message Handlers (ASH) were similar concept of application-defined code in the kernel network stack that could also interact with the application [87]. Plexus was a similar mechanism of the Modula-3 SPIN operating system for user-defined functions in the kernel. New protocols could be implemented per application and still interface with the other OS components [39]. The SPINE extension brought application code in Modula-3 to the NIC, e.g., for dumping video data directly to the frame buffer or implementing an IP router on the NIC [88].

The SHRIMP project explored virtual memory mapped NICs as RDMA mechanism for cluster computing [89]. In contrast to Ethernet the sender determines the target address. The role of the kernel shrinked with more support from the hardware which otherwise needs to be emulated. System calls still existed but mapping more registers and sharing the state with applications would have further reduced the kernels role.

The Virtual Interface Architecture specification (VIA) was an attempt to standardize the memory layout of virtual NICs both for RDMA as well as regular message exchange [90, 91, 92]. Adoption was low but the paravirtualized virtio NICs of recent hypervisors can be seen as late successors.

The *x*-kernel protocol framework offered a flexible network stack with a modular architecture for protocol layering [93].

Splitting the kernel's TCP into a `SOCK_DGRAM` datagram part in kernel space for demultiplexing and a userspace part for the actual TCP logic was explored to, e.g., support innovation for TCP by decoupling it from the kernel [94].

The Exokernel was an OS design that facilitated direct hardware access [95]. Instead of providing file and socket abstractions to processes the small kernel gave shared access to disks and network devices. DPF, a packet filter with dynamic code generation, demultiplexed the incoming packets [96]. The applications on top were library operating systems which, e.g., implemented the POSIX interface as library. This way a web server could directly send the prepared IP responses for its files from a disk cache. The scheduler accepted predicate expressions to check whether a sleeping process is runnable. This converted a polling model into a blocking model with less context switches [97].

Nemesis was a *vertical structured* operating system with similar principles [98]. Applications were running as library operating systems and the kernel's task was reduced to scheduling and resource accounting of the virtualized hardware. While Nemesis was initially based on ATM, Ethernet, and IP was added later with demultiplexing in the device driver [99].

The Barrelfish multikernel OS targets heterogeneous NUMA cores. It features application-level networking based on the lwIP TCP/IP stack [100] and uses a special user-level RPC (URPC) for communication with other networking processes [101].

The GNU HURD microkernel OS has a concept of subhurds which are similar to nestable Linux containers. The difference is that they do not share one network stack because it runs as separate userspace process per subhurd. Access to the system services for TCP/IP and for Ethernet devices is based on translator nodes in the virtual filesystem which allows redirection. This way one can choose between a port of the Linux TCP/IP stack and lwIP [102], and even layer two network stacks for a VPN. Also the Ethernet may either be accessed through a direct driver or a L2 switch service [103].

The Genode OS Framework is a tool for building microkernel operating systems. Similar to GNU HURD it provides a POSIX interface for C applications where the network stack is a separate process accessed via a virtual filesystem. Applications may share one network stack process or use a separate one. Currently the choice is between a port of the Linux TCP/IP stack and lwIP. Each network stack needs its own IP because demultiplexing takes place at the IP layer [104].

2.9 Packet Access and Switching with the Linux Kernel

Operating systems with BSD sockets allow implementation of userspace protocols or monitoring traffic with RAW sockets. RAW sockets get a copy of every incoming packet and can directly transmit L2 packets. Programmable in-kernel filters, as present in the initial Xerox Alto packet filter, the Berkley Packet Filter (BPF) [105, 106] in UNIX/BSDs and its Linux variant, improve efficiency by reducing copies of unwanted packets to a RAW socket. Instead of the usual singe-packet copies for each syscall, the `PACKET_MMAP` option on Linux gives userspace programs access to a memory-mapped ring buffer for zero-copy batched packet transfers. RAW sockets are most interesting for protocols that are unknown to the kernel because otherwise interference of the two network stacks requires firewall rules. Daytona was a port of the Linux TCP/IP stack to userspace working on RAW sockets with firewall rules and dummy sockets allocated on the kernel stack [107]. Alpine used the unmodified FreeBSD network stack in userspace on RAW sockets [108].

Common operating systems also support TAP devices with distinct MACs and IPs. A TAP device is a virtual interface in the kernel that communicates with a userspace process via Ethernet frames. The process behind the TAP device forwards the packets to a virtualized guest OS NIC or performs userspace networking. The host kernel usually acts as L3 router, NAT, or as L2 bridge to connect the process behind the TAP device to the internet. With additional iptables rules the kernel can also forward NAT ports to TAP devices to share one IP for multiple services. A special use case is VPN encryption where the kernel routes its traffic to the TAP device which itself uses the kernel network stack to communicate to a VPN server. The Xen hypervisor are also creates virtual interfaces in the dom0 host kernel. Similar to TAP devices the host kernel either bridges or routes the packets to the NIC.

For unprivileged user VMs that can not set up TAP devices, the QEMU hypervisor uses a Slirp-based NAPT (Network Address Port Translation) device backend that parses TCP/UDP traffic and forwards it through TCP/UDP sockets of the host kernel stack.

Linux containers often use a pair of connected virtual interfaces (veth) with one present in the container networking namespace and one in the host networking namespace. The veth endpoint in the host networking namespace is usually connected to the outside with an extra bridge device via IP routing or L2 switching. To reduce configuration overhead, macvlan devices can replace the veth device and the additional bridge at the host. By assigning multiple MACs directly to the NIC with limited L2 switching in the kernel, macvlan devices are more lightweight but lose communication with the host kernel. The similar macvtap devices offer this functionality for userspace networking or virtual machines. In the passthru mode a single macvtap device takes over a NIC. For networks that do not permit multiple MACs per NIC but multiple IPs, ipvlan and ipvtap are virtual devices that route the IP traffic.

The traffic control system of the Linux kernel supports actions with eBPF bytecode on packet ingress and egress for filtering and redirection [109]. It acts earlier than the netfilter/iptables system.

The new Linux XDP project [110] uses the eBPF bytecode VM for early packet processing in the NIC driver before the kernel network stack will process the packets. Possible actions for a packet are dropping, emitting a modified packet, and forwarding it to the kernel network stack, another CPU or NIC, or to a userspace socket. In contrast to the traffic control system, XDP does not work for egress. While XDP will likely supersede other firewall mechanisms in the kernel [10] it also brings switching abilities for containers, VMs, and userspace networking. With an appropriate eBPF bytecode supplied to the kernel driver a userspace network stack is able to match packets on their ports and claim them for special AF_XDP sockets [111, 112]. Depending on XDP support in the driver either a zero-copy driver mode, a copy driver mode, or a generic mode exist. For an untrusted application it is required to steer the traffic to a separate NIC queue for the zero-copy mode but using another mode lifts this requirement. An AF_XDP socket achieves the performance of kernel bypass with DPDK if the NIC driver implements zero-copy XDP forwarding. An example for XDP usage is Cilium, a container security and load-balacing middle-ware [113], and an example for AF_XDP socket usage is the OVS-eBPF patch for Open vSwitch [114].

2.10 Kernel Bypass Solutions for Linux

The third-party kernel module PF_RING provides a syscall interface for direct packet access so that the kernel's network stack will not receive the packets anymore [115]. Without patched drivers and the commercial license only some functionality is available.

Netmap also provides direct access to the NIC rings in a standardized way [73]. The kernel does not receive packets anymore but netmap exposes the kernel's NIC rings. An application can decide to forward traffic for the kernel's network stack by putting unclaimed packets into the kernel's NIC rings. While the netmap syscall interface is part of FreeBSD Linux does not include it by default. The kernel module and its patched drivers are maintained separately from the Linux kernel.

PFQ is a kernel bypass framework with a functional Haskell-like domain specific language for in-kernel execution [116, 117]. Similar to XDP, with `AF_XDP` a user-supplied isolated in-kernel filter program may decide to the drop a packet, broadcast, or forward it to userspace endpoints or the kernel network stack, or leave the decision to the next filter program. PFQ supports up to 64 userspace endpoints per NIC. The kernel module works with unmodified drivers and offers automatized patching for improved performance.

Small UIO kernel modules allow for userspace drivers to access the NIC via a memory mapped region and character device for interrupt notification. The DPDK library [76] uses this mechanism to implement its polling mode drivers in userspace. While DPDK is the most notable kernel bypass framework with a userspace driver, others are Snabb for Lua and the educative ixy ixgbe driver in C, Rust, Go, and C#.

2.11 Software Switches for Linux

The last sections already introduced PFQ and XDP. They rely on the appropriate eBPF or pfq-lang code to forward packets to, e.g., a userspace networking application.

Open vSwitch (OVS) [118] is a programmable multilayer software switch written in C for VMs on Linux KVM/Xen hypervisors and others. It may be distributed over multiple servers. On Linux it uses a kernel module but relies on userspace components to set up rules. With DPDK it also supports kernel bypass. Similar to load-balancing, multiple network services can share one IP.

BESS (previously SoftNIC) is not a switch itself but a modular framework to build switches or network functions by combining the C++ modules with a Python script [119]. It supports VM clients with vhost-user virtio NICs and also containers via a special kernel module for VPort virtual interfaces.

VMs with QEMU/KVM often use a paravirtualized multi-queue virtio NIC on a TAP device. There is still a copy of the Ethernet frames from the virtual guest NIC buffers to the TAP device. Instead of relying on the userspace QEMU process to do the copy, the KVM vhost-net kernel module can directly switch between the virtio packet buffers and the TAP device. The vhost client implementation in QEMU is, however, not tied to the kernel module. This enables vhost-user as a mechanism to share the virtio guest NIC buffers with a userspace switch such as Open vSwitch on DPDK or Snabb.

With userspace NIC drivers there is no copy to the kernel involved. SV3 is a similar userspace switch which also implements its own drivers for the host NIC [120]. The DPDK-based NetVM switch does not use virtio but defines an own shared memory model for VMs to further reduce copy overheads [121].

In contrast to userspace switches, the netmap framework includes VALE/mSwitch as an in-kernel switch for netmap endpoints [122]. It provides virtual endpoints and connects them as L2 bridge with the NIC and optionally the kernel network stack. Passthrough of netmap endpoints to VMs reduces overheads of virtualized guest NICs for netmap-enabled applications. VALE can be extended with additional kernel modules. One of them is VALE-bpf which itself is programmable with eBPF bytecode [123].

The VALE-based HyperNF is a NFV framework for fair resource usage and implemented as Xen modification for hypervisor-based IO [124]. With HyperNF the VALE switch in the dom0 Linux instance exports data structures for Xen in order to perform packet forwarding without a VM context switch to dom0. Thus, HyperNF improves on ClickOS which uses VALE in the dom0 Linux instance to forward packets to the VMs [125].

The MultiStack module for VALE differs from the other switches and is rather meant for userspace processes than VMs. In order to share one IP with the kernel it lets userspace network stacks register the ports they want to receive [126]. MultiStack requires the process to issue a special syscall to receive packets via a netmap interface. MultiStack does the necessary packet parsing in C as a kernel module.

Swarm is a userspace switch which similar to MultiStack allows multiple userspace networking applications to share one IP by registering their ports [127]. Swarm accesses the NIC via netmap or macvtap but does not forward packets to the kernel's network stack. It uses the shmif shared memory bus interface [128] for IPC with the userspace networking applications because the example applications use the NetBSD's TCP/IP stack as rump kernel.

2.12 Hardware Support for Packet Demultiplexing

The research gigabit Ethernet NIC Arsenic [129] had hardware support for exposing virtual interfaces to applications. Sending and receiving happened without involving the kernel. The kernel installed filters on the NIC which determined the virtual interface for each packet and the packet header offsets in order to place headers and data in distinct memory areas. The NIC also validated outgoing packets and did QoS accounting. As demonstration the Linux TCP/IP stack was ported to userspace.

Commercial Solarflare NICs provide kernel bypass with the OpenOnload framework [130]. A wrapper script for network services overwrites redirects the socket syscalls to OpenOnload. OpenOnload's userspace TCP/IP stack sets up hardware port matching rules for a hidden NIC queue to bypass the kernel.

Multi-queue NICs typically do receive side scaling (RSS) with a hash on the protocol headers. Advanced NICs support steering with programmable filters. This allows isolation of a application packet flow in one NIC queue. The DPDK-based TAPM manages dedicated *hwTAP* receive and transmit queues per userspace network stack through a hardware filter for protocol-destination tuples [131]. Each userspace network stack can then register which tuples to receive and the host kernel stack is also stays available. Netmap also supports to work on only a specified queue. This reduces the need of software matching to forward packets to the host kernel network stack. The problems with this approach are the limited number of queues and the task of keeping the queues distinct. This will also break streams with IP fragmentation because the header info is missing unless reassembly took place on the NIC.

Some recent NICs support hardware-assisted virtualization (SR-IOV) by exposing virtual interfaces as separate PCI devices. The kernel can unbind the PCI devices for VM passthrough or userspace IO. A management software offloads filter rules to the NIC to forward the packets to the desired virtual interface usually with the same mechanism as for RSS queues. The Arrakis research operating system, for example, gives each application a virtual interface but uses separate MACs instead of matching on ports due to insufficient matching rules [132].

2.13 APIs of Network Stacks in Userspace

A network stack in userspace can provide an alternative socket API to increase performance gains. Another value is backwards compatibility by providing similar BSD socket API as the kernel. Using the same API requires only small source code changes. No source code changes are required with library preloading at runtime with the LD_PRELOAD environment variable. LD_PRELOAD gives higher priority to the userspace networking library over those of the libc system call wrappers, thus, redirecting the API. The mentioned OpenOnload is one example for backwards compatibility through redirection.

LOS is a DPDK userspace networking daemon which runs a shared networking stack [133]. It also relies on LD_PRELOAD to redirect socket calls of applications. For syscalls such as `read` and `write` it distinguishes between the kernel's file descriptors for and the userspace socket file descriptors by allocating its own ones from $2^{31} - 1$ downwards while the kernel counts upwards from 3.

The maintained and feature-rich network stacks of Linux and FreeBSD network stacks are often targets for userspace ports. The LibOS NUSE project or the LKL project run Linux kernel code as

library on top of TAP interfaces, netmap, or DPDK [134]. No recompilation for existing applications is required with LD_PRELOAD.

Examples for FreeBSD are libuinet [135] on netmap or F-Stack on top of DPDK [136] which require source code changes. The mTCP stack resembles the BSD socket API but requires source code changes [137]. It supports netmap and DPDK as backends. Sandstorm is a userspace network stack on netmap for closer application integration to reduce copy overheads [138]. It is not ported from a kernel TCP/IP stack and has a different socket API. SeaStar is ScyllaDB's userspace network stack based on DPDK with an API for asynchronous futures and promises [139]. The lwIP network stack mainly aims for embedded devices but also runs on POSIX userspace with an additional custom blocking API [100].

Using the NetBSD TCP/IP code as Rump kernel library in userspace on Linux was done by reimplementing missing some parts such as clock interrupts. Applications that include the Rump network stack must either use LD_PRELOAD or change the source code to call the userspace socket functions instead of the libc functions.

As an alternative to API compatibility through LD_PRELOAD, FINS replaced the AF_INET protocol family of the Android Linux kernel with an own kernel module that relays the all socket syscalls to a userspace implementation [140].

Picoprocesses and DUNE processes maintain a hybrid position between userspace networking and (unikernel) virtual machines. POSIX applications can run in a picoprocess on Windows that emulates the system calls through binary rewriting to relay the networking syscalls to lwIP [141]. IX is a framework introducing an intermediate layer between the host kernel and the application by running as DUNE process on the KVM hypervisor [142, 143]. DUNE processes do not implement a full kernel but relay system calls to the host kernel via hypercalls. Yet IX handles networking system calls in the DUNE process with lwIP on top of DPDK. It also introduced new batched syscalls. By porting a common IO event library to these new system calls, legacy applications based on the event library need not to be ported. With IX, the network stack is protected from the application flaws but still is not memory safe and has access to the host kernel. ZygOS improves on IX with a work-stealing architecture, RSS, and giving up on run-to-completion [144].

3 Analysis

The first section describes a thread model and a set of requirements for memory-safe networking on Linux specially when multiple memory-safe network services and the host kernel network stack share one IP. The section also explains the implications. The second section gives proposals for the requirement of protecting network stacks without memory safety. The third section compares operating system VMs, unikernel VMs, and userspace networking as architectures for memory-safe networking. The fourth section has a detailed look on how to realize direct packet access according to the set of requirements.

3.1 Threat Model and Requirements for Memory-safe TCP/IP

I will assume the following threat model regarding the capabilities of an attacker. The attacker uses an Internet endpoint device and is not an ISP. The attacker⋯

- has access to knowledge about a vulnerability in our Linux kernel network stack
- can craft malicious IP headers, L4 headers, and payloads sent to our service
- cannot control the Ethernet frames sent to our service
- cannot spoof its source IP address for packets sent to our service
- cannot monitor unrelated packets from our service

Based on this threat model I will now discuss requirements that the deployment of multiple memory-safe network service should fulfill to guarantee that untrusted packets are handled by a memory-safe component on the system. The requirements have to cover the following different settings.

The memory-safe network service can be the only network stack on a system and no other network stacks are exposed to the Internet. The first case is a unikernel running on bare-metal. However, a unikernel may lack the many needed drivers to deploy a memory-safe network service in a general setting without targeting a certain hardware. Linux is the most general operating system for servers and embedded devices. Hypervisors, such as Xen, rely on Linux for hardware support. Therefore,

the second case is a memory-safe network service with a dedicated NIC running as process or VM on Linux, given that it is the only part exposed to the Internet.

The second case is much more likely, leading to my assumption that Linux will be involved regardless whether unikernel VMs or userspace networking provide the memory-safe networking with any mean for NIC access. The network stack of the host Linux kernel often has to stay active for administrative tasks, time synchronization, and system updates, which require network access. To not undermine the gains of memory-safe networking, the kernel's network stack should be protected from untrusted input.

Dedicated NICs are not generalizable and tie the number of services to the number of NICs. A memory-safe network service may have to share the NIC through L2, L3, or L4 switching with other network stacks, which may or may not be memory-safe. To keep the gains of memory-safe networking, a L3 or L4 switch should be memory-safe.

The threat model assumed that the attacker is a regular client and only controls the TCP/IP headers but not the Ethernet header. This means that a memory-safe handling of Ethernet frames is not required. The NIC driver and a L2 switch can form a TCB under the assumption that the NIC and the Linux kernel correctly process differently sized Ethernet frames as generated by the preceding router. Therefore, the NIC driver and any L2 handling do not have to be memory safe and the large amount of supported NICs on Linux stays at hand.

In conclusion the following requirements should be considered to deploy a memory-safe network service on Linux:

- Allow only memory-safe TCP/IP processing for untrusted packets, e.g., in L4 switch, firewall, packet monitor
- Allow TCP/IP processing without memory safety only for trusted packets, e.g., packets for the kernel network stack must be filtered, with trust defined by a policy or heuristic

The first requirement does not apply to dedicated NICs or L2 switching of memory-safe network stacks. Just protecting any network stacks without memory safety would be required by only exposing them to trusted packets. If no network stacks without memory safety are connected to the Internet it is not relevant.

However, L2 switching needs own MAC and IP addresses for each service, probably resolved by an own DNS entry for public services. Besides the scarcity of IPv4 addresses it introduces more configuration and administration overhead. For VMs many processes share one MAC address but for userspace networking an IP-per-process scheme means that a multiprocess architecture needs to share the packet access interface via IPC or become a multithread architecture. There are settings like Wireless LANs where only one MAC is available but multiple IPs which need memory-safe L3 switching or Proxy ARP. If only one IP is available, having L2 switching requires memory-safe

NAT with internal IPs. Since memory-safe networking already is a deviation it should ideally not introduce many hurdles and difference in comparison to using the kernel's network stack where many network services share one IP. While the separation on L2 or L3 for independent network stacks ensures that there are no port clashes, it is still possible to share one IP and forward the ports to different network stacks even without NAT. They just should not listen on the same port or establish connections with the same local ports to the same endpoint. Besides the reasons to favor a generalizable approach for network services it also suits client systems to share one IP for memory-safe networking, e.g., in a web browser. According to the requirements, memory-safe L4 switching is needed to share one IP for multiple network stacks. Any network stacks without memory safety, such as the kernel network stack, have to be protected.

3.2 Protecting Network Stacks without Memory Safety

The requirements stated that only trusted packets should be processed by a network stack without memory safety. In most cases this will be kernel network stack. Trust for an incoming packet needs to be defined by a policy or a heuristic.

A simple solution is a memory-safe firewall filter which has the policy to only let packets through from trusted connections. Thus, a packet is trusted if the connection is trusted. I propose to treat response packets from outgoing connections of the kernel network stack as trustable. The additional assumptions are that the communication partner is not compromised, no MITM attacks take place such as DNS manipulation, and there is no way for an attacker to forge response packets through IP spoofing and surveillance of the needed header information. I also propose to treat ARP packets are trustable because they have a local origin.

Another solution would be to filter packets based on a heuristic which rejects malicious packets. Intrusion detection/prevention systems, such as Snort, try to achieve this by matching against a set of rules. Whether a packet is malicious is hard to know because it would already require knowledge of bugs in the network stack. A possible heuristic, however, would be to treat valid packets as less harmful even though there is no guarantee that the kernel network stack does not have bugs for valid packets. Invalid packets are more likely to cause exceptional states with a malicious intent. Rejecting invalid packets does not have negative effect but in practice the memory-safe packet validation does not have the same features as Linux kernel stack. An example are less used protocols such as SCTP, or certain IP options or TCP options which would need to be rejected because the following packets can not be validated. Also it is a questionable overhead to track the TCP state of every connection to reject packets mismatching the current state. The rustwall component for seL4 [65] as introduced in the section on Rust networking follows the main idea of this alternative solution. Anyway this solution should be used in addition to the first one which defines trust for full packets because the heuristic does not say whether the application payload is malicious. An

application using the kernel's network stack such as system tools are likely not implemented in a memory-safe language.

A stronger variant of the heuristic for kernel network stack protection would be to rewrite all incoming packets by reassembling the payloads in a memory-safe network stack and construct new packets for forwarding. It has the same effect as the filter heuristic because invalid packets are not forwarded. In addition it removes attacker direct control on the packet headers. Yet it suffers the same disadvantage of lacking feature parity with the kernel network stack. Overall, such a rewriting of incoming packets should only be seen as additional protection besides trusting communication partners because it does not consider whether the application payload is malicious.

If the kernel network stack does not receive ICMP error messages anymore because they may be malicious, it has to use path MTU discovery over TCP. This is already common in network settings where a firewall blocks ICMP messages. Incoming ICMP echo requests, known as pings, should be handled by a memory-safe network stack.

3.3 Memory-Safe Network Services: OS, Unikernel, or Process

This chapter analyzes the solutions for memory-safe network services which were introduced in the last chapter. A subsection on unikernels describes the state of unikernels in general and their relation to userspace networking. Another subsection about memory-safe userspace network stacks points out the current limitations and needs.

Some memory-safe network stacks or operating systems are rather historical examples. I will not consider them here.

I also do not consider Biscuit OS and gVisor due to the following reasons. Biscuit OS is written in Go and one would need to prove that the network stack is not suspect to memory corruption due to data races. Also gVisor is written in Go, the implementation of the Linux ABI itself running in Linux userspace. It can be seen as a paravirtualized VM because it reuses functionality of Linux via syscalls. It is not yet a complete ABI reimplementation but could be used to for a memory-safe network service. With Rust, for example, using the standard library sockets means that the libc will issue socket syscalls to gVisor. The Go network stack of gVisor relies on RAW sockets for L2 access on veth devices which need own IPs. This scheme for L2 access means that the host kernel network stack handles the incoming packets and dispatches them to the raw sockets. To fulfill the requirements of the previous section gVisor would need direct L2 access without IP routing to RAW sockets. Nevertheless the issue of memory corruption due to data races persists and either proving

of absence is required or compiling gVisor with the race detector, further slowing down the low performance due to RAW sockets.

The current solutions for memory-safe network services are:

- MirageOS/HaLVM unikernel VMs for Haskell/OCaml applications
- RedoxOS VMs for Rust applications
- Userspace networking on TAP devices for Haskell, OCaml, or Rust

MirageOS uses mirage-tcpip as network stack in OCaml which also supports userspace networking on Linux with TAP devices. HaLVM's network stack HaNS in Haskell also works on Linux with TAP devices. RedoxOS as a microkernel in Rust relays the networking to a userspace service based on the network stack smoltcp. As the others, smoltcp works on Linux with TAP devices. All three TCP/IP implementations are actively maintained. Using them for userspace networking will be discussed in its own subsection.

RedoxOS has the advantage of being a multiprocess OS and supporting the Rust standard library. Since it is a microkernel it relays the networking syscalls to a userspace service based on the network stack smoltcp. For many Rust applications this is a convenient way towards memory-safe networking. However, RedoxOS is still in its early days and is not compatible with Linux and requires porting of applications.

Unikernels with MirageOS are the most prominent option due to the interest in unikernels but still they have to be considered a niche. One reason might be the choice of OCaml as functional language. Another reason might be that the unikernel architecture differs much from programming a network service as process on a general purpose OS. The number of available libraries is limited, no processes can be spawned, and the common debugging tools for processes do not apply as unikernels run in CPU Ring 0. While for HaLVM there is experimental multi core support, MirageOS only supports cooperative lightweight threads. There is no loopback interface which the VMs could use to communicate with each other by expecting a service on a certain port. Instead, they need knowledge of the respective IP of, e.g., a local database server. The database would also be accessible via the IP which may not be wanted and requires blocking outside access. Also for end users the puristic approach of immutable unikernel images with configuration at compile time means that they also have to compile the OCaml codebase with the appropriate configuration. Memory-safe unikernel frameworks achieve the goal of memory-safe network services by focusing on the essential set of features for a microservice architecture.

3.3.1 Unikernels: Features and TCB

A unikernel architecture comes with many drawbacks. This subsection looks at the properties related to supporting existing software, fast network access, and the TCB due to the hypervisor. These determine the value of unikernel VMs in relation to userspace networking.

The mentioned memory-safe unikernels do only support a limited scope of OCaml and Haskell programs. Closer to supporting existing software written for Linux are unikernel frameworks not written in a memory-safe language. In contrast to MirageOS and HaLVM projects such as Rumprun and OS^V provide a POSIX-like interface but of course no forking or spawning of processes. Multithread support ranges from cooperative multithreading to pthread and multicore support. Still, the lack of feature parity with Linux and the required maintenance work led to the initiation of the UKL project. With UKL the Linux kernel's code base should be used as unikernel, linked with application code and a glibc backend for direct function calls into the kernel [145].

Since unikernels usually do not run on bare metal but on a hypervisor with Linux, it is questionable if the VM abstraction is the correct choice since unikernels do not need the separation of CPU Rings and operate in a single address space. Indeed instead of using KVM hardware virtualization it was possible to modify the Solo5 unikernel base code and the ukvm/hvt unikernel tender to spawn processes by directly mapping the hypercalls to syscalls on Linux, restricted with a seccomp policy [146]. Among the many benefits such as dynamic memory allocation and increased performance this reduced the calls to involved Linux kernel code, implying a smaller attack surface.

The ukvm/hvt runtime for unikernel VMs on KVM requires multiple context switches per packet from and to TAP devices. By adding support for the netmap kernel bypass API the less context switches and packet batching improved the TCP performance of MirageOS [147].

For memory-safe network services the unikernel architecture is mainly enabling the implementation of OS primitives in a memory-safe language, while still relying on a TCB of hypervisor and Linux kernel for device drivers. A userspace process in a memory-safe language, however, usually takes the kernel as TCB, even though it is not required to make use of, e.g., the kernel's filesystem and one can reduce syscall usage to a minimum. With userspace networking the direct exposure to malicious input can be handled in a memory-safe language. It depends on the network service implementation if it gives direct or indirect control over, e.g., filesystem syscalls to an attacker. If the attacker cannot influence syscalls in a critical way, then memory-safe unikernels and processes are not a big tradeoff in terms of security and other factors such as ease of development/use would favor a process model.

3.3.2 Memory-safe Userspace Networking

Userspace networking on Linux with TAP devices in OCaml, Haskell, or Rust is mainly aimed for development purposes. It could also serve as deployment model and provides the benefits of the large Linux ecosystem. A process can spawn other processes or connect to the Linux loopback interfaces, e.g., for data base access. Another advantage of userspace networking is that no VMs are involved, thus, regular client programs such as browsers or the Tor P2P anonymity layer can also benefit from memory-safe networking.

None of the current memory-safe userspace network stacks integrates well with the kernel loopback interface. If the userspace network stack does not provide this transparently but local communication is needed, changes in the application logic would be required to additionally use the kernel sockets.

Other major drawbacks are that TAP devices are slow and need to be bridged on L2 and have outside IPs to be meaningful for memory-safe network service deployment. Userspace networking with kernel bypass is popular for performance but its usefulness for memory-safe TCP/IP was not utilized yet. One big hurdle is how to gain L2 packet access while sharing one IP. The other is having a userspace networking library which does not require changes in the application logic and is easy to use.

Both the mirage-tcpip network stack and the HaNS network stack provide socket functions which allow the application to compose statements in parallel threads.

With somltcp, however, the socket functions are non-blocking and can only used in a single thread. For parallel actions on sockets the main loop needs to track the state of each connection and try the read/write actions for each socket. At the end of the main loop it needs to wait for arrival of new packets or a timer event, and perform packet ingress/egress of the sockets. This scheme does not fit to Rust applications which maybe do not have a suitable main loop and rather need to compose statements of socket operations. Many network projects in Rust directly use the standard library TCP sockets with blocking IO. Others use the asynchronous TCP socket types from the Tokio IO event loop. Tokio itself is based on the mio non-blocking IO library which uses the standard library sockets together with `epoll` event notification. Since smoltcp does not provide blocking IO similar to the standard library or asynchronous IO for Tokio, a network service cannot easily be ported to use smoltcp for userspace networking.

Userspace networking for memory-safe network services in Rust is promising if a transition path without heavy source changes exists for regular applications. Out of the different ways for providing userspace networking as presented in Section 2.13 one decision has to be made on either decoupling the choice of the network stack from the source code and even programming language

(choice at runtime), or having stronger safety guarantees and optimizations with source code or build configuration changes (choice at compile time).

Choice at runtime through the LD_PRELOAD environment variable a userspace network stack works for even programs in C and overwrites the libc syscalls with function calls for the userspace network stack. Direct syscalls, however, are not covered because they are rarely used. Disadvantage is the possible confusion of Linux kernel file descriptors and those of the userspace network stack sockets. Also the behavior of the libc and the Linux kernel must be exactly remodeled and all functionality implemented not to break applications. This interface via the C calling convention can not be proven memory-safe by the Rust compiler and type mismatch may happen. A more obscure alternative would be patching the AF_INET packet family in Linux through a kernel module, so that socket calls of all programs on the system are redirected to the userspace network stack.

Choice at compile time can be done with source code changes such as different import statements to use the socket types of a userspace networking library. This allows for partial usage of userspace networking when the Linux kernel has to be used for certain operations. Still, the userspace networking sockets should detect if loopback traffic requires usage of kernel sockets. Ideally source code changes are not necessary if a build flag for a dependency switches to userspace networking. This would be possible if, e.g., the mio library had a feature flag for userspace networking, and setting it is enough for an application that uses Tokio or mio sockets. The big advantage of compile time choice is that the network stack and the application are more integrated. The application can use special zero-copy functions and benefit from code inlining optimizations. This approach also provides stronger safety guarantees by the compiler which can check for type mismatch and if a function for a socket is implemented or not. Feature completeness of all standard library functions is not necessary as long as those that are used are implemented.

3.4 Dedicated NICs, L2 and L4 Switches

If dedicating a NIC is affordable for a memory-safe userspace network service, DPDK, netmap, and macvtap in passthru mode can take over a NIC. One big difference for kernel bypass frameworks is whether the NIC driver remains in the kernel or is reimplemented in userspace. Under the considerations of Section 3.1 memory-safe drivers are not needed. The kernel has drivers for the majority of NICs which is an advantage for macvtap and netmap. Yet for unpatched drivers the netmap performance is not good and not all NICs have patches. The optimized C userspace drivers of DPDK run in polling mode which is an advantage for high-performance systems but otherwise hurts energy efficiency.

A L2 switch can share the NIC for multiple memory-safe network stacks and the host network kernel but requires assigning different IPs. The VALE switch [148] exposes a netmap interface on

virtual ports with global names. Attaching to the switch requires the userspace processes to have the needed privileges to access the virtual `/dev/netmap` file. VALE connects the userspace network stacks and the host kernel network stack. Therefore, packets for the memory-safe network stacks are not processed by the kernel's network stack. Memory-safe networking with MirageOS can be deployed on VALE if built for the custom solo5-netmap ukvm target.

While VALE only targets netmap applications, a L2 bridge in Linux can switch TAP devices which are common for userspace network stacks as well as VMs. Suitable bridges are the kernel virtual bridge devices, Open vSwitch, and the bridging functionality of macvtap. The bridge configuration has different variations with and without connectivity to the host kernel stack, an own IP of the bridge interface, and possibly IP routing. It is still less complex than an Open vSwitch configuration. The most simple solution is macvtap in bridge mode because it does not need a separate bridge interface. Macvtap devices are directly configured as sub-interface of the NIC and have a low overhead because there is a fixed set of MACs to switch on. However, connectivity to the kernel network stack is lost unless an extra macvlan device is configured for the kernel. Memory-safe networking with MirageOS, HaLVM, or RedoxOS can be deployed on a bridged TAP/macvtap.

Since bridging with Linux kernel can involve with L3/L4 parsing for the flow dissector, traffic control actions, and firewall rules, compared to VALE the attack surface is larger. If the kernel network stack stays exposed with an own IP it needs protection according to the suggestions in Section 3.1, such as a memory-safe firewall which only forwards responses for outgoing connections. TAP devices need one syscall per packet while VALE's netmap API supports batched packets. After a TAP device is created it can be assigned to a user. This loosens the need of special process privileges.

The most general approach is L4 switching to share one IP. It does not require a pool of additional IPs and a static configuration to assign them. It makes dynamic attaching of userspace network stacks possible for short-lived service instances or regular client programs such as browsers. Userspace networking applications have to attach themselves to the switch and register their open ports. The following L4 switch solutions were developed for this purpose:

- MultiStack [126] VALE module for netmap applications
- TAPM [131] NIC queue management service for a modified lwIP TCP/IP stack
- swarm [127] service for rump kernel interfaces and NetBSD rump kernel TCP/IP stack

MultiStack is a VALE module and, thus, exposes virtual netmap interfaces. Besides using these interfaces, the application must allocate dummy kernel sockets and issue special `ioctl` syscalls to bind the ports. MultiStack forwards all packets with other ports to the kernel network stack. Packet matching is done in the kernel module with C. MirageOS unikernels can currently not work on MultiStack because the solo5-netmap ukvm target does not register the open ports.

TAPM requires hardware support in the NIC to manage separate queues for the kernel network stack and each userspace network stack. Packet matching is not done in software but in hardware.

The userspace switch swarm take over a NIC and does not forward packets to the kernel network stack. If swarm is used on top of VALE it would be possible to have the kernel network stack active with another IP. Packet matching is done in userspace with C.

None of the three solutions fulfills the considered requirements for memory-safe userspace networking, i.e., memory-safe packet matching and protection of the kernel network stack. They give direct L2 packet access for userspace network stacks without passing them trough the kernel network stack, but lack in the following points. MultiStack and TAPM forward all packets directly to the kernel network stack if the packets are not for the registered ports. This exposes the kernel to malicious packets and would need a protection scheme as mentioned previously. MultiStack and swarm parse the packets not in a memory-safe language. The TAPM source code is not published and, thus, cannot be used in practice to develop memory-safe networking on top of it. An additional challenge is IP fragmentation where a simple port matching works only for the first packet and will discard all others. IP reassembly is needed to handle these corner cases. It is not possible for TAPM to add this as hardware rule. For MultiStack and swarm it implies to add a lot of C code, increasing the possibility of memory-corruption bugs.

Using a L4 switch to share one IP for userspace networking should ideally not come with limitations compared to L2 switching. Endpoints with different IPs are bridged on a L2 switch and the same way a L4 switch should also allow, e.g., connections from VMs to userspace networking, regardless if the VM is bridged by the kernel network stack or directly attached to the L4 switch. Swarm running on VALE partly addresses this but also requires the kernel network stack to have a different IP than the userspace network stacks. TAPM switches with matching on the NIC and, therefore, needs an additional hardware switch in VEPA hairpin mode to connect endpoints with different IPs.

Connectivity between endpoints of the switch that share one IP cannot be provided by the switch because a network stack will not send out packets to its own IP address. Userspace network stacks have to use the kernel loopback interface for these connections.

3.5 Building Blocks for a Memory-safe L4 Switch

This section will analyze the possible building parts to design a L4 switch for memory-safe userspace networking. Some are just techniques for L2 packet access or forwarding packets to the kernel network stack, others are full programmable switch frameworks.

Netmap has all features for a userspace L4 switch in a memory-safe language. It even supports zero-copy forwarding to the kernel NIC rings. The switch can implement a firewall by sniffing on outgoing connections and only allowing response packets to be forwarded to the kernel. Netmap pipes for fast IPC to endpoints also use the same netmap API. They support zero-copy forwarding if access is granted to the shared memory areas of switch and kernel. Shared memory access is

not critical for an own memory-safe networking service but for untrusted or exploitable programs. To open netmap devices endpoints need access to `/dev/netmap` which implies full control over netmap. Instead of configuring group/user permissions for this file for unprivileged endpoints, a handover of the file descriptor is also possible. If the NIC supports the necessary matching rules to separate hardware queues, userspace network stacks can directly use these queues and no software matching is needed. This, however, would break IP fragmentation, the ability to connect endpoints with different IPs with each other, and sniffing on outgoing packets. The major disadvantage of netmap is the need of compiling the kernel module and patched drivers for the target kernel, which can fail if the kernel is too new for netmap.

VALE is distributed with netmap and supports custom modules running in the kernel. The VALE-bpf module enables to program the switching logic with isolated eBPF bytecode [123]. The L4 switch could be implemented as eBPF program in C. Even though C is used, remote code execution is not possible with eBPF. With a shared data structure the userspace network stacks could control the matching rules. The VALE-bpf module has to be build for the target kernel.

The PFQ kernel module is a framework for programmable switching in a functional language. Each endpoint can provide a small program to match its own packets. The endpoints are, however, not connectable with each other and no actions are defined on packet transmission. An additional component to firewall the kernel would need to be programmed. PFQ would also require a userspace helper to update the rules by monitoring outgoing packets with a RAW socket. Similar to netmap/-VALE this kernel module needs compilation for the target machine and automatized driver patching for full performance.

DPDK brings its own userspace drivers for popular NICs. A L4 switch can use it for direct NIC access but the kernel does not see the interface anymore. DPDK Kernel NIC Interface (KNI) is a kernel module which allows the switch to forward packets to a virtual interface. While forwarding the switch can also enforce firewalling. This virtual interface can have the same IP and routing table entries as the NIC had before DPDK took over. Userspace network stacks could be connected to the switch with a IPC mechanism for packet exchange, e.g., Unix Domain Sockets or virtio-user interfaces. DPDK does not include as many NIC drivers as the kernel.

The BESS modular switch framework is implemented in C++ and would need reimplementation of the L3/L4 handling in a memory-safe language.

With the Snabb framework the switch can be implemented in Lua. It provides Lua NIC drivers and virtio-user interfaces for userspace networking endpoints. Similar to DPDK it does not support as many NICs as the kernel. The kernel also needs a virtual interface since the NIC driver moves to userspace.

Macvtap works with any NIC and does not need third-party kernel modules. A L4 switch in userspace can use macvtap in passthru mode to access all packets from a NIC. The kernel will not

receive packets anymore but still has the interface configured. Kernel forwarding can be solved with an additional TAP device as virtual interface for the kernel with higher priority in the routing table (a technique common for VPNs). The TAP device should use the IP of the original interface which itself gets a placeholder IP. The TAP device is connected to the switch which can do the firewalling for kernel protection. The connection of the userspace network stacks to the switch needs an IPC mechanism.

The newest Linux kernels include AF_XDP. An eBPF program for XDP acts on a single hardware queue and can forward packets either to a userspace endpoints or the kernel network stack. Same as PFQ XDP only works for incoming packets and cannot connect endpoints with each other. A workaround would be to only forward the kernel's packets with eBPF and all others to a userspace switch, and from there use another IPC mechanism as connection for the endpoints. To monitor outgoing kernel traffic to update the firewall rules though a shared data structure, either a userspace helper with a RAW socket is needed or a traffic control eBPF egress action.

4 Design

This chapter describes the proposed design for a memory-safe L4 switch in userspace that improves over previous solutions not meant for memory-safe networking. The switch fulfills the set of requirements which was considered for deploying memory-safe network services on Linux. It allows sharing one IP for memory-safe userspace networking and the host kernel while preserving the benefits of memory-safety through firewalling.

In addition a proposed library for memory-safe userspace networking in Rust which takes advantage of the switch is described. It provides the same socket API as the standard library. The behavior of programs does not change since they keep the same IP as the kernel and can reach the local interface.

4.1 usnetd: A Memory-safe L4 Switch in Userspace

The design of the memory-safe L4 switch needs to cover NIC access, forwarding packets to userspace endpoints as well as the kernel network stack, registration of open ports, packet matching rules, and firewall rules for the kernel. Ideally, it should also allow dynamic creation of endpoints for unprivileged processes and also allow for inter-endpoint traffic with different IPs (e.g., from a VM device in the kernel network stack to a userspace networking service).

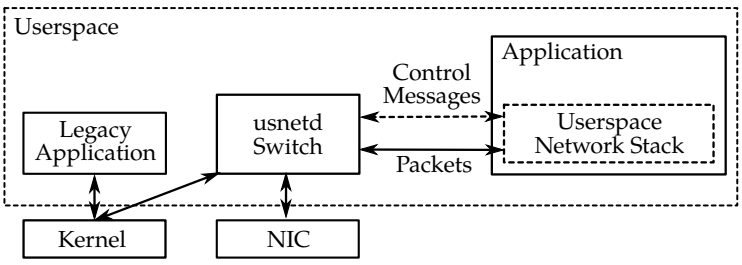

Figure 4.1: *usnetd*: Memory-safe L4 switch in userspace

I will present a model for such a switch in userspace and refer to it as *usnetd*. Its relation to userspace network stack, kernel network stack, and legacy applications is shown in Figure 4.1. The model consists of a frontend for userspace network stacks and a backend for NIC and kernel packet forwarding. It does not rely on a particular kernel bypass framework but instead has several variants

of NIC access. The switching logic is handled in userspace, I decided against a loadable in-kernel bytecode for VALE-bpf, PFQ, or XDP.

The VALE module VALE-bpf would provide all needed features to implement the switch in C compiled to isolated eBPF bytecode. The userspace network stacks need to organize which open ports and VALE interfaces each one claims through a shared data structure with the eBPF code in the kernel. However, deciding on VALE-bpf would mean that netmap is a hard requirement, limiting the usefulness of the switch to certain supported kernel versions and NICs.

PFQ as programmable switch does not provide all needed features. Each userspace endpoint would register a pfq-lang script to match their ports on packet reception. However, PFQ only supports 64 endpoints. In contrast to VALE-bpf PFQ does not act on packet transmission, thus, cannot connect endpoints with each other if they have different IPs, nor monitor ports of outgoing packets as needed for the firewalling of the kernel.

AF_XDP sockets with eBPF programs as a switch have similar problems because they lack egress actions to monitor the endpoint outgoing packets of an endpoint and redirecting them in order to connect different IPs. With AF_XDP sockets as userspace endpoints, each endpoint should have its own NIC queue with hardware packet matching and an eBPF program to either redirect packets to the userspace network stack or pass it further to the kernel network stack. Since the number of hardware queues is limited, the queue's eBPF program can also redirect packets to multiple userspace network stacks on this NIC queue. To firewall the kernel, all queues should be covered. This static allocation of all queues, however, makes it more complicated work with dynamically created AF_XDP endpoints. Each endpoint would need to update the matching rules in the shared eBPF data structure for open ports itself. As workaround to at least monitor outgoing packets of the kernel, traffic control eBPF actions are needed to update the matching rules of the shared eBPF data structure.

By choosing to handle all switching logic at a central component in userspace, usnetd can support multiple NIC backends and there is no hard dependency on a certain kernel module. This design can also cover different IPC mechanisms for the endpoints. The switch should handle all traffic, between kernel and NIC, userspace network stacks and NIC, and kernel and userspace network stacks. This enables to monitor outgoing packets and connect endpoints with each other.

4.1.1 usnetd: NIC Backends

This subsection presents possible NIC backends for the switch. Some include forwarding capabilities to the host network stack. Others include special IPC channels that can be used for zero-copy forwarding to endpoints.

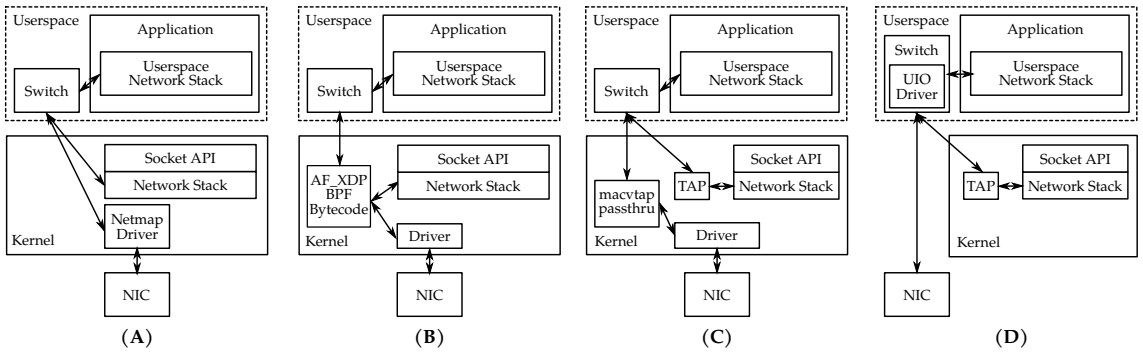

Figure 4.2: Different variants for a usnetd NIC backend: *(A)* netmap, *(B)* AF_XDP, *(C)* macvtap, *(D)* UIO driver, such as DPDK, in userspace

Figure 4.2 shows the possible variants for the NIC backend, labeled from *(A)* to *(D)*. In variant *(A)* with netmap the NIC interface stays active in the kernel, thus, no configuration of a virtual interface is needed. Netmap exposes the same simple syscall API for NIC rings, kernel rings, and netmap pipes as virtual interfaces for the userspace network stacks. Other benefits are packet batching and zero-copy forwarding. Since netmap pipes are used for endpoint IPC, MirageOS unikernels VMs with the solo5-netmap patch can be endpoints as well. Even though netmap is not included in mainline Linux and requires building the kernel module, it is a solid base for a prototype implementation.

Variant *(B)* uses AF_XDP just for NIC access for the switch. The userspace network stacks would need an IPC mechanism that is able to connect to the switch. Unix domain sockets are a simple solution while custom shared memory areas or virtio-user can give higher performance. AF_XDP needs eBPF code to redirect packets to either the switch or the kernel, with dynamic rules for open userspace ports and firewalling of the kernel. This implies to set up AF_XDP sockets for all NIC queues and monitoring the outgoing kernel packets via additional eBPF code in the traffic control subsystem. This complexity brings the benefit of directly working on newer kernels without third-party kernel modules. Besides being very new, the disadvantages make AF_XDP not a good candidate for a prototype implementation of usnetd.

Instead of forwarding to the kernel with eBPF code and monitoring outgoing packets with another eBPF code, the switch could also claim all traffic. Since the kernel needs connectivity, the switch would set up a virtual TAP interface to be the primary interface for the kernel, configured similar to the original one which gets retired. This is same as the kernel forwarding part for variant *(C)*.

Variant *(C)* works on old and new mainline kernels by claiming all NIC packets with a macvtap device in passthru mode. Since the kernel cannot use the interface anymore, an additional virtual TAP interface has to be configured, replacing the original interface which is unusable. Using macvtap is less complex than AF_XDP but has a lower performance. The secondary macvtap device and the additional TAP interface for the kernel with higher-priority routing table entries make managing the interfaces a bit more complex. The connection to userspace network stacks can be provided by

a custom IPC mechanism. Macvtap is a good base for a prototype but the workaround with the TAP interface and changed routing metrics may causes problems in real use cases.

In variant *(D)* the switch completely takes the NIC from the kernel through a UIO driver in userspace, e.g., with DPDK. Since there the interface is no present anymore, the kernel needs a new virtual interface. It can be either a slower TAP device or a faster DPDK KNI device provided by a separate kernel module. Using DPDK is more suited for a high-performance setting with dedicated cores, usage of 2 MB or 1 GB huge pages and NIC polling. Here again, the connection to userspace network stacks needs a custom IPC mechanism. DPDK is muss less universally usable than macvtap but the fact that the original NIC disappears has slight advantages over macvtap. DPDK is more complex to use than netmap and, therefore, not a good base for a prototype.

4.1.2 usnetd: Interaction with Userspace Network Stacks

The separation of backend and frontend simplifies the interface for endpoints. It hides the complexity how the management of all forwarding rules is implemented. For dynamic endpoint creation the switch needs a system-wide control socket so that userspace network stacks can attach to it. A protocol of control messages defines the allowed interaction for setting up forwarding rules.

The switch should separate IPC channels for control messages from IPC channels for packet exchange. Regardless if the creation requires privileges, the switch can create such a channel and hand in over to the userspace network stack. A Unix domain socket as control socket supports handing over of file descriptors, e.g., for netmap pipes or Unix domain sockets. Static endpoints could also be supported with a switch configuration. Through messages to the control socket, a userspace network stack can dynamically request an IPC channel and register open ports for it as needed.

Which NICs the switch uses should be decided at startup. A userspace network stack has to request an IPC channel per underlying NIC. This simplifies the switch logic because otherwise it would be required to extract routing information from the kernel to forward an IP packet to the correct NIC. The userspace network stack has to decide which NIC it uses for a connection. The interfaces of the kernel network stack that are connected to the switch have the same principle and are only sending out on one NIC.

In all presented variants the packet transfer goes through the switch. By observing outgoing connections, the switch can add matching rules for incoming packets. This is required for firewalling the kernel network stack according to the previous considerations for trusted packets. The switch can apply the same principle to userspace network stacks and handle them the same way as the kernel network stack by adding rules for outgoing connections. Only listening ports need to be announced to the switch, which simplifies porting of a userspace network stack to work together

with the switch. With an additional static configuration for open ports in the switch, no changes are needed for, e.g., a MirageOS unikernel VM. This is not possible with MultiStack, TAPM, and swarm which always require the process to register its open ports.

To match for response packets for outgoing connections, the switch matching rules have to not only include the local port but also the remote IP and remote port, looking for packets matching the 5-tuple `(localAddr, localPort, remoteAddr, remotePort, ipProtocol)`. If two network stacks establish an outgoing connection from the same local port to the same remote endpoint a port clash will happen. To prevent this it is advised that userspace network stacks register a match for local IP, local port, remote IP, and remote port. They could also query the switch for free local ports before sending out the first packet but this includes the possibility of races. To prevent the kernel to use local ports which clash with userspace network stacks, the userspace network stacks should not use the range of the kernel's local ports as configured in `/proc/sys/net/ipv4/ip_local_port_range`.

Trusting response packets for outgoing kernel connections is a simple protection mechanism. It is only effective if the attacker cannot monitor outgoing packets and spoof its IP address to create a malicious response. It also assumes that the connection partner is trustworthy and not compromised because malicious payloads may harm the application.

Other additional heuristics such as packet header validation can be applied to further protect the kernel. It would be best to reduce usage of the kernel network stack by providing a memory-safe userspace network stack for all applications through LD_PRELOAD with a libc-compatible library, or a memory-safe local SOCKS proxy.

The matching rules for open listening ports include the IP protocol type, e.g., UDP or TCP, and the IP address. The userspace network stack has to register the rule which can fail with an error message if the port is already used by another userspace network stack. Since the kernel should not have exposed open ports there is no clash. However, the userspace network stacks should still allocate kernel sockets to enable local communication with other network stacks. Using the same port for the kernel sockets would fail. The matching rule includes the IP address which means that the endpoints are not forced to share one IP. This supports VMs as endpoints. Specifying ports is optional for a match rule, so that IP protocols without ports work as well, e.g., for receiving ICMP echo responses.

Incoming ICMP packets which are not responses to outgoing ICMP echo requests have to be handled differently since the switch does not create forwarding rules as with expected responses for outgoing ICMP echo requests. To answer ICMP echo requests a dedicated memory-safe userspace network stack could register itself to receive any incoming ICMP packets that were not matched already. The switch could also answer them directly. However, incoming ICMP error messages due to a dropped packet at a remote host should be forwarded to the network stack that sent the dropped packet. In

order to find the correct network stack the switch needs to parse the ICMP packet contents. The switch could also mirror the ICMP error messages to any memory-safe network stack. Untrusted ICMP packets should not be sent to the kernel network stack and, therefore, the kernel needs to use path MTU discovery over TCP as if it was behind a restrictive firewall which blocks all ICMP traffic. By dropping any incoming ICMP error messages in the switch instead of finding the correct sender network stack, the userspace network stacks are in the same situation as the kernel. This would serve as simple solution which is sufficient for a TCP service with path MTU discovery.

The switch should generate ICMP error messages or TCP RST packets in response to incoming packets if it cannot forward a packet to any network stack. However, this is not strictly required because not sending out response packets for closed ports is a common technique against port scans. A userspace network stack could also register itself to receive any IP packets that were not matched already and send out ICMP error messages as response.

The switch uses the same matching rules for incoming and outgoing connections by making the remote IP and remote port an optional part: `(localAddr, localPort, optionalRemoteAddr, optionalRemotePort, ipProtocol)`. First it looks for matching rules which include the remote IP and remote port set, then it looks for a rule where they are unset. This way a userspace network stack can also register a rule for a single incoming connection from a remote IP and remote port. Use cases are a migration of a connection from a userspace endpoint with a general listening socket to another userspace endpoint for DDoS isolation. Another scenario is manually registering a trusted remote IP for incoming connections to the kernel.

Since ARP packets have a local origin they are trusted and the switch broadcasts them to all userspace endpoints, the kernel, and the NIC. To connect userspace network stacks and VMs with different IPs, the switch should recognize local MACs used as destination and match rules as for incoming packets.

The following algorithm implements the required packet actions:

1. If outgoing packet: Note the source MAC as a local MAC
2. If ARP packet: Broadcast to all other endpoints, end here.
3. If outgoing packet not from a listening address: Add match rule for response packets
4. If incoming packet from NIC, or if outgoing and the destination MAC is a local MAC:
    a) If matching rule found for `(localAddr, localPort, remoteAddr, remotePort, ipProtocol)`: Forward to endpoint, end here.
    b) If matching rule found for `(localAddr, localPort, any, any, ipProtocol)`: Forward to endpoint, end here.
5. If outgoing packet with non-local destination MAC: Forward to NIC, end here.

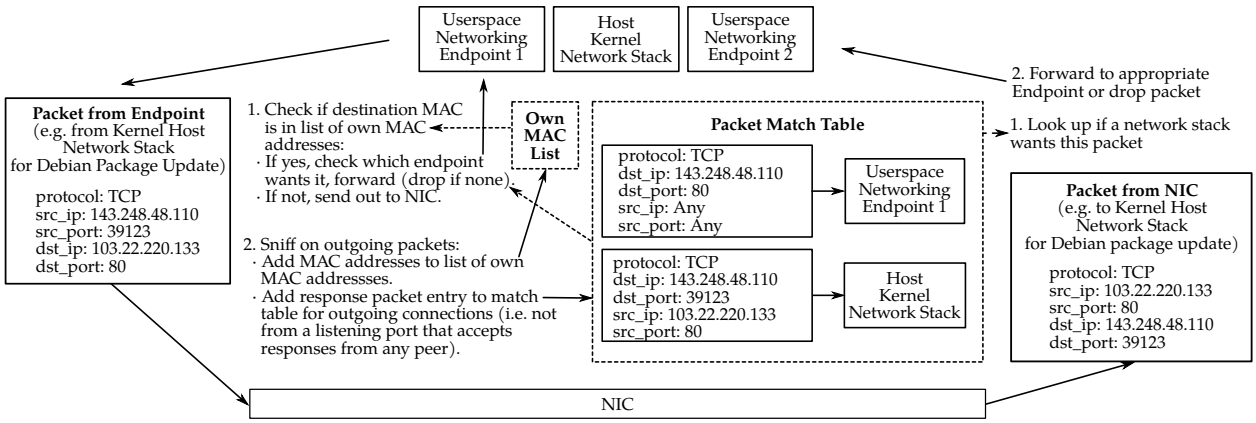

Figure 4.3: Switch packet forwarding logic, with example entries for memory-safe web server on 143.248.48.110:80 and an outgoing connection of the kernel network stack (Debian update package HTTP download)

The example in Figure 4.3 illustrates this matching logic with two rules. One is for a memory-safe userspace web server on 143.248.48.110:80 and one for an outgoing connection of the kernel network stack initiated from local port 39123 to 103.22.220.133:80 (Debian update package HTTP download). It shows an outgoing packet and incoming packet both from the kernel network stack.

If a userspace network stack registers a matching rule for an open port, it should deregister it again if it does not use it anymore. The switch still needs to remove stale rules not only because it automatically adds new ones for outgoing connections but also because a userspace network stack may terminate without deregistration through a control message. This problem is linked to detecting if a userspace network stack is still alive or the stale IPC channels should be removed. The userspace network stack should delete itself from the switch with a control message so that the switch removes the IPC channel and all related rules. This, however, does not work for a forcefully terminated program. For Unix domain sockets as IPC mechanism, a write error occurs for lost connection and the switch can clean up this endpoint as soon as forwarding fails. Netmap pipes, however, have no mean to detect if the endpoint is still receiving packets on this channel. Also the automatically added outgoing rules should be cleaned up regularly. Therefore, the switch should check if for rules of the kernel network stack actually sockets exist in the system. Userspace endpoints should tell their PID to the switch when they request an IPC channel, so that the switch can check if the owner of a channel is still alive. These cleanups should at least happen before the switch adds new endpoints or rules because a restarted service can otherwise not reuse its old port.

In conclusion the control messages in Table 4.1 form the protocol between userspace network stacks and the switch. The protocol includes messages for IPC channel setup for both packet exchange with Unix domain sockets and netmap pipes, but it could be extended with a custom shared memory channel. Switch and userspace network stacks communicate with control messages on a Unix domain socket. With a Unix domain socket as control socket special syscalls can hand over file descriptors in addition to a message. Authentication only happens according to the control socket's

Table 4.1: Control Messages from userspace network stack to usnetd

Control Message	Answer
`RequestNetmapPipe(Interface, PID)`	nmreq struct (and file descriptor handover)
`RequestUDS(Interface, PID)`	"$" (and file descriptor handover)
`AddMatch(IP, Protocol, [Port], [Source IP], [Source Port])`	"OK"/"ER"
`RemoveMatch(IP, Protocol, [Port], [Source IP], [Source Port])`	-
`QueryUsedPorts`	`QueryUsedPortsAnswer( listening: (ipProtocol, localIP, localPort)…, connected: (ipProtocol, localIP, localPort)…)`
`DeleteClient`	-

file permissions. The switch trusts the userspace network stacks and does not need to check against the following possible misbehavior: Claiming all ports so that new connections cannot find free ports anymore, or hijacking traffic by registering rules with specific remote IPs at the same local port as another service.

4.2 usnet_sockets: A Rust Userspace Networking Library

A memory-safe userspace networking library for Rust should provide a TCP/IP implementation accessible through a socket API. To easily add userspace networking to an existing Rust code base, the API should stay the same regardless if userspace networking or the Rust standard library is used. Through usnetd userspace TCP/IP can keep the behavior of running as process on a common operating system such as Linux and sharing the IP with the kernel. It should also interact with other programs through the loopback interface.

A strong dependency on usnetd should not be required for cases which want to dedicate a NIC or use L2 switching with separate IPs. With a runtime configuration the library should allow to use one of macvtap, netmap, or usnetd for NIC access. If possible the library should automatize interface setup such as creation of a macvtap device.

The configuration can either specify a NIC explicitly or take the NIC of the kernel's default route. For each NIC the access method can be one of usnetd, macvtap, or netmap. The configuration should include IP and MAC addresses. If the userspace network stack shares the IP with the kernel or takes over the full NIC, it can copy the IP and MAC of the interface. For multiple IPs on one NIC with usnetd, the same MAC can be used. The L2 switches in macvtap and netmap's VALE, however, require a different MAC and IP. A new MAC can be random or static. A new IP needs either a static

assignment with subnet and gateway, or a configuration through DHCP. The name of a netmap device may refer to a NIC, a NIC queue, a netmap pipe, or a VALE port. To find the correct MTU the configuration requires the name of the underlying NIC because the netmap buffer size can be larger than the underlying MTU.

The user should provide a combination of the following configuration options as environment variable, either set globally or for single programs:

- usnetd:
  - interface: `$NAME`, or discover from default route
  - IPC channel type: netmap pipe, or Unix domain socket
  - MAC: copy from interface, use random, or `$ADDR`
  - IP: copy from interface, use DHCP, or `$IP $SUBNET $GATEWAY`
- netmap:
  - interface: `$NETMAPNAME` with MTU of `$INTERFACE`, or discover from default route
  - MAC: copy from interface, use random, or `$ADDR`
  - IP: copy from interface, use DHCP, or `$IP $SUBNET $GATEWAY`
- macvtap:
  - interface: create for `$NAME`, or existing `$MACVTAPDEV`
  - MAC: copy from interface (takes over NIC), use random, or `$ADDR`
  - IP: copy from interface (take over NIC), use DHCP, or `$IP $SUBNET $GATEWAY`

If the library has multiple interfaces configured it should use them according to the kernel's routing configuration. Connections to local IPs, such as the loopback or interface IPs, should use kernel sockets for loopback communication. Also sockets bound to only a loopback IP should only use the kernel sockets. Other sockets bound to a specific interface IP or any IP should use both the userspace network stack and a kernel socket. Otherwise there would be no connection between programs using the kernel network stack and those using the userspace network stack.

There are two approaches for providing the same API as the Rust standard library. Either the userspace networking library provides the same types through a Rust library and requires changed import statements, or it provides the same ABI as the libc or Linux syscalls through interception at runtime. Both approaches do not exclude each other and the library can support both. Another design decision is executing the network stack as an external process or within the application process.

I decided on an in-app network stack in the form of a Rust library. This allows for a partial application of userspace networking, code inlining, and compile time checking of the interface between application and library code. This way the library can start small and only implement the functionality most application need. Unsupported functions lead to a compile-time error. Providing the

same ABI as the libc is a huge task and involves managing of the different file descriptors. It also needs unsafe Rust code to handle the memory buffer arguments. Still, if the implementation reaches full feature parity to provide a libc ABI, it can serve as language-independent wrapper library for memory-safe userspace networking. It has low priority because the first goal is to support Rust applications.

Many Rust applications use the socket types of the Tokio event loop or the underlying non-blocking IO library mio. To support these applications without source code changes, the Rust mio non-blocking IO library needs to be ported to the userspace networking library. The applications just need to be complied with a directive to use a different mio branch.

The socket types in the Rust standard library follow the behavior of Linux syscalls and do not rework the API towards idiomatic Rust patterns. An example is returning that "0"bytes were read for a closed connection instead of returning an enumerate type which encodes the end of the steam. Also in contrast to the usual ownership semantics, the standard library supports cloning of sockets, resulting in concurrent access to the kernel sockets.

The userspace networking library should behave in the same way and allow moving of socket types to other threads and cloning them. Concurrent access requires mutual exclusion so that, e.g., reading from a socket does not return the same data twice. The library should also count how many cloned socket type instances exist and only close and deconstruct the underlying socket if the last socket type instance closes the socket.

The socket types of the Rust standard library also expose their underlying raw file descriptors. The file descriptors can be used with for custom `select` or `poll` syscalls. With unsafe constructor calls even a new socket can be created from a socket's file descriptor. Userspace sockets, however, do not have file descriptors, and returning some other file descriptor can silently break existing unsafe code. Therefore, if existing code with `select`, `poll`, and file descriptor conversion should use the userspace networking library with minimal changes, a special type has to be provided to emulate file descriptors. This special type would not be usable as file descriptor and only work together with special `select`, `poll`, and conversion calls provided by the library. The special `select` and `poll` calls would be wrappers around the real ones and should allow to either use real file descriptors or the emulated ones from userspace sockets.

Hostname resolution on UNIX systems uses the Name Service Switch subsystem which specifies the order of querying files such as `/etc/hosts`, local MDNS, a special library, or DNS servers. To cover all this behavior the hostname resolution can rely on glibc and use the kernel network stack for communication. In addition, configuring a local memory-safe DNS resolver with userspace networking as DNS server helps to keep the kernel network stack communication internal and allows to use the glibc hostname resolution in Rust. Otherwise without replacing the local resolver

and if only DNS needs to be resolved, the userspace networking library could provide trait similar to `std::net::ToSocketAddrs` based on a Rust DNS client library.

The kernel network stack copies the data between kernel network stack buffers and the userspace socket calls which the Rust standard library reflects. Therefore, the userspace networking library should also provide copying reads and writes between network stack and application. An additional function, however, for zero-copy reads and writes directly on the socket buffers is possible.

An application may have longer computations without any calls to the socket types. Without any background thread the network stack would miss packet answer timeouts and emptying the queue of incoming packets which leads to packet drop. Therefore, the library needs a background thread to provide good TCP performance for busy applications. Without a background thread the notification system for blocked socket calls would be more complicated with all of them either sleeping or polling their state. Nevertheless, a single-thread application which ensures that it has frequent socket calls, can run without the background thread for a better performance.

The userspace networking socket types should use one network stack for the whole application and configure it according to the environment variable. For specific purposes as well as testing of the library it is useful to expose types that allow manual construction of a network stack and its sockets independently from the global state.

In summary, applications can easily be ported because the library provides similar socket types as the standard library. Partly porting of an application is also supported. Userspace sockets integrate well with other programs that use kernel sockets.

5 Prototype

This chapter describes the prototypes for usnetd and usnet_sockets which realize the previous design. The implementation choices were not primarily oriented on performance but demonstrating the idea. As prototypes they do not yet cover all the functionality needed to replace the Linux network stack in every setting.

5.1 Implementation of usnetd on netmap

The prototype of the usnetd switch is implemented in Rust. This allows for code reuse with the userspace networking library. I decided to implement NIC access, kernel forwarding, and IPC channels with netmap which is variant *(A)* of Figure 4.2. The other variants with macvtap, `AF_XDP` or DPDK could be added with a choice for one of them at startup.

Which NIC the switch should use is provided as environment variable or through a configuration file as program argument. The global control socket is created at `/run/usnetd.socket` as Unix domain socket in datagram mode. Only a specified user group has access because the switch gives direct packet access to normal processes which they usually do not have. Userspace network stacks should talk to the control socket from an own unique bound Unix domain socket which serves as identification. The control messages use JSON for serialization. The userspace network stack can either requests a netmap pipe as IPC channel or a pair of Unix domain sockets. The file descriptor for one side of the channel is handed over to the userspace network stack.

The switch maintains a global list of all endpoints connected to the switch. Endpoints are the userspace network stack IPC channels, the NIC, and the NIC rings of the kernel network stack. Both the NIC rings of the kernel network stack and the userspace IPC channels have noted for which NIC they are. Figure 5.1 shows an illustration of the components and their interaction.

In the main loop the switch waits for input events from all file descriptors at the same time. File descriptors are used for the netmap interfaces for NIC, kernel NIC rings, userspace network stack IPC channels, and the control socket. For each endpoint with new packets all are forwarded before the next endpoint is processed. I did not yet explore parallel forwarding because netmap zero-copy forwarding to multiple interfaces with batching through implicit TX when polling was much easier

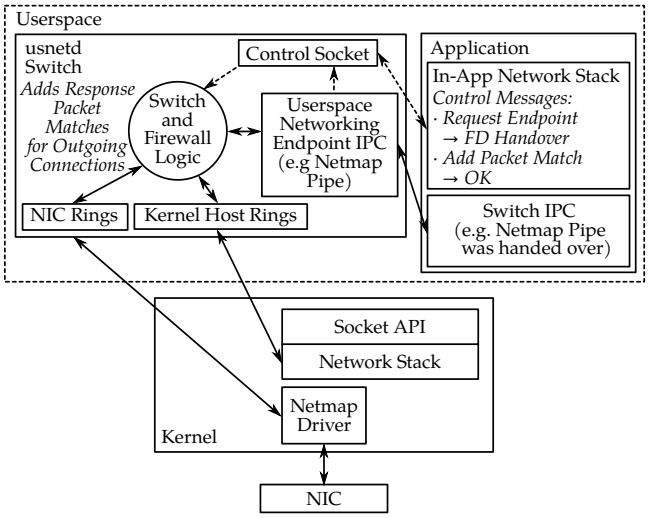

Figure 5.1: The switch *usnetd* on netmap and interaction with a memory-safe network service

to organize in a single thread. The forwarding logic follows the algorithm in subsection 4.1.2. Port numbers only apply for TCP, UDP, DCCP, SCTP, and UDP-Lite. For other protocols only rules which match any port apply. A global hash table holds the mapping of wanted packets to endpoints. Local MACs are stored in a global list. The listening ports of an endpoint are noted in the endpoint in addition to the packet match in the global hash table. This spares going through the hash table to prevent adding rules for outgoing packets that come from a listening port.

The main loop also processes control messages for new matching rules. Removal and creation of new endpoints is deferred until the end of the main loop where packet forwarding has finished. This is rather and implementation detail to solve the problem of iterator invalidation because the iterated list of endpoints should not change during iteration. Only when endpoints are removed or created, the switch also checks for stale IPC pipes without an endpoint process and cleans them up first. This is accomplished by a global list of tuples of PIDs and endpoints. The switch also cleans up all rules of a removed endpoint.

The switch uses device types that are compatible with smoltcp so that the code for netmap access is shared with the userspace networking library. I implemented reading a packet from netmap rings with directly proceeding the buffer pointer instead of waiting until the packet is forwarded. The rationale was to keep the netmap code simple because it consists of unsafe blocks where the Rust compiler cannot reason about memory safety. This means that all netmap interaction should be kept simple and short because it is in the trusted code base in contrast to L3 and L4 handling. Directly proceeding the pointer means that the packet must be forwarded before waiting for new packets from file descriptor since this waiting hands the buffer pointer back to netmap. This is not a hurdle for future concurrent packet matching but means that there should only be one global waiting action for the file descriptor as barrier for all threads. Only proceeding the pointer after the forwarding has taken place means to require access to both the NIC and the endpoint at the same

time which would introduce locking into concurrent packet matching which is not desirable. A more sophisticated solution would be managing of the ring entries in categories of *forwarding* and *forwarded* is needed to decide on which entries can be handed back to netmap. An easier option which allows to directly swap the entries for forwarding with free ones and proceed the ring as usual would be having a pool of free entries by using a dummy netmap pipe as intermediate place.

The netmap device types have an additional function for zero-copy forwarding to another netmap device. Zero-copy forwarding works between NIC, kernel NIC rings, and the userspace networking stacks that use netmap pipes as IPC channel. My initial implementation of the necessary unsafe Rust code had a wrong pointer assignment which lead the netmap buffer to be used by netmap as well as the network stack. I could only notice this through double free error messages from the netmap kernel module and occasional failed Rust assertions due to memory corruption. Even after I solved this bug I kept an option to disable zero-copy forwarding for the switch. The usual packet reads and writes in netmap rings also involve unsafe Rust code, and all syscalls for polling and Unix domain sockets need unsafe Rust as well. As mentioned, this does not violate the requirement for memory-safe TCP/IP services because L2 handling and the NIC drivers form the trusted code base.

The netmap pipes have numbered identifiers and usnetd manages the pool of 4096 possible identifiers when it creates a pipe for handover to a userspace network stack. The switch creates pipes in the shared memory area of the netmap NIC rings and the kernel host NIC rings. This is not mandatory in general but necessary for zero-copy forwarding. It is also one of the reasons why by default the access to the switch is limited to a trusted user group. If the system has no untrusted processes this restriction can be lifted.

Packet transmission with netmap is implicitly done during the syscall that waits for new packets at the beginning of the main loop. This saves final transmission (TX) syscalls per netmap file descriptor but comes with the problem that the TX ring is not always updated. Therefore, as workaround if no TX buffers are found, a transmission syscall is used to update the state of the TX ring.

The switch does not generate ICMP error messages for closed ports and does not answer ICMP echo requests. These parts currently have to be handled by a userspace network stack which needs to register broad matching rules for these cases.

This current implementation of the usnetd switch includes support for PCAP dumps of all seen packets. However, it lacks some features that the design demanded. The switch does not clean up packet rules of the kernel outgoing connections. It only detects IPv4 packets and ARP but not yet IP fragmentation, IPv6, multicast, and broadcast packets. Also implementing support for NIC access with macvtap and a virtual TAP interface for the kernel as designed in switch variant *(C)* would be a good fallback option if netmap is not available.

5.2 Implementation of usnet_sockets with smoltcp

I decided to base the Rust userspace networking library on the TCP/IP implementation smoltcp. As a network stack for embedded devices the smoltcp API does not resemble the standard library API. Smoltcp has no blocking calls and instead in a main loop the application should do the possible socket operations, wait for new packets or timer actions if necessary, and do socket egress/ingress. The read and write calls in smoltcp can directly operate on the socket buffers through a closure function provided by the application.

The behavior of the standard library with blocking calls was modeled on top of the smoltcp primitives. Therefore, I decided against any modification to smoltcp so that the library just uses it as a dependency.

At the time I started the library the user base of smoltcp was slightly growing and it got integrated into RedoxOS. Nevertheless, I had many bugs to fix until TCP transfer was reliable on a 10G NIC. The bugs I discovered were rarely causing runtime panics but rather prevented the transmission from going forward, resulting in stalled connections. This included logical bugs in the TCP state machine regarding the retransmission state and timer, fast retransmit state, and remembering the lastly announced window. In one case the transmission was even closed too early. Arithmetical overflow errors in the sequence number calculation caused an array slice to be of the wrong size. Another array management error was present in the packet assembler and in the calculation of the TCP option length. After these were fixed reliable transfer of multiple GB/s was possible.

To let the user choose from various methods for NIC access, I implemented additional smoltcp device types. Supported are macvtap, netmap devices, and usnetd IPC channels with handover of netmap pipes or Unix domain sockets. The library constructs them at runtime according to an environment variable with the interface configuration as planned in the design.

The library exports `TcpStream` and `TcpListener` types which provide already most of the functionality that the standard library provides. They are owned by application threads and have handles to the underlying smoltcp sockets. A background thread is spawned to exchange packets from the NIC and smoltcp sockets. It wakes up blocked socket type calls in application threads to let them check if they can continue. Since smoltcp currently is limited to single-thread usage, the background thread and the socket type calls in the application threads share access to smoltcp sockets via a mutex. The background thread waits until either new packets arrive or packet timers expire. If the background thread waits for incoming packets but the socket type call in the application thread produced data for transmission, it will break the waiting of the background thread through writing to a special file descriptor so that the background thread transmits the data to the NIC. Figure 5.2 shows current solution with a global mutex to lock the socket set.

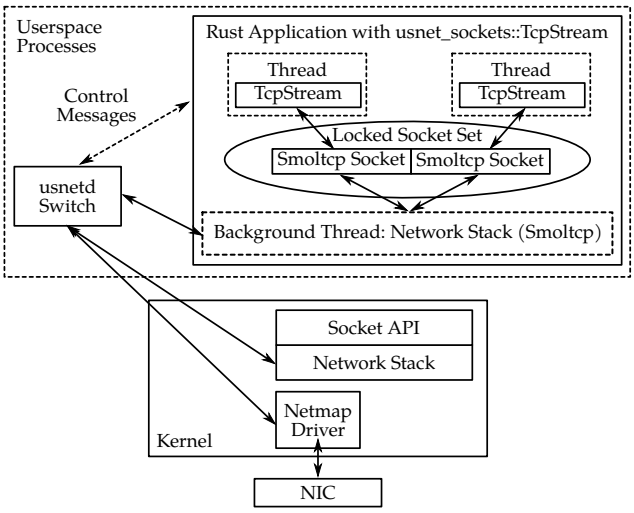

Figure 5.2: The Rust userspace networking library *usnet_sockets* around smoltcp

Building a correct model of the kernel socket behavior with smoltcp primitives was not straightforward. Connecting a socket, reading, and writing all differ in terms of when they have to wait for incoming data and when they have to force transmission. For example, a `read` call can change the window size from a zero window to a non-zero window which needs transmission of a packet to notify the sender that it can continue sending. Also the return code of a `read` has to be `0` if the socket closed which translates to checking that the smoltcp socket cannot receive anymore. In addition to these common functions as defined by the standard library, I also implemented a special zero-copy API with closures to work on the socket buffers. It exposes this functionality of smoltcp to applications which want to spare the copy of the read and write calls. However, smoltcp will always copy the incoming packet payloads to the socket buffers.

I also implemented the integration of the userspace network stack with the kernel network stack by attaching to the kernel's loopback interface. The library's socket types choose transparently to use userspace networking or the kernel's loopback interface for a connection, and also listen on both. For the `accept` call the background thread not only waits for incoming connections at the userspace networking interface but also for incoming connections on the file descriptor of the loopback kernel socket.

The library exposes a multi- and a single-thread version of the API and the default can be chosen with a build flag. The single-thread version has no background thread and no locks. It directly performs the work of the background thread in place during a socket type call. Therefore, it is only suited for single-thread usage of an application which mainly calls into the socket API and does no long computations.

Environment variables configure the socket buffer size for receive window scaling and the number of backlog sockets for parallel incoming connections. A variable also allows to pin the background threat to a CPU core.

The current implementation lacks in the following points. The wake-up notification system with the global lock around smoltcp needs to be replaced with a better parallel packet processing and socket access. This requires changes to smoltcp for read/write locks for the sockets and per-socket notifications. More pressing are, however, the missing features both in the library prototype as in smoltcp. In smoltcp these are congestion control algorithms, Nagle's algorithm, selective ACKs, delayed ACKs, IP fragmentation and reassembly, DHCP, IPv4 options, and more such as ICMP details. The library does not consider UDP, IPv6, multiple IPs for `bind` and `connect` calls, emulation of raw file descriptors and `select` or `poll` syscalls, multiple NICs and routing, non-blocking calls, configuring timeouts and TTLs, and path MTU discovery. A workaround for MTU reduction is implemented through an environment variable.

6 Evaluation

This chapter presents performance measurements of the prototypes for the usnetd switch and the usnet_sockets userspace networking library. It also demonstrates adding userspace networking support for a HTTP library in Rust.

The tests measure performance between two directly connected machines with the same hardware specification. Both test machines run the same software configuration. This means that either both machines use direct NIC access with netmap, or both use macvtap passthru, or both use usnetd on netmap with netmap pipes.

The test machines are two directly connected Intel Xeon X5550 servers with a CPU frequency of 2.6 GHz. In addition some tests also use two connected Intel Core i5-4690 desktops with a CPU frequency of 3.9 GHz. The higher single-core performance is the main difference. To expose the weaknesses of the prototypes only the slower machines are used in some experiments. All machines use Intel 10G NICs with a patched ixgbe driver for netmap. Ethernet flow control is disabled to ensure highest transmission speeds even if the receiver drops packets. The processes are pinned to a core for more consistent and favoring results.

The packet processing speed for the switch will be measured in Mpps (million packets per second). The highest Mpps value is expected for the smallest packet size because the larger the packets are the fewer can be transmitted as they already saturate the link. For minimal-sized Ethernet frames 14.8 Mpps saturate a 10G link while for maximal-sized frames already 0.8 Mpps saturate the link. A packet processing speed less than 14.8 Mpps means that the throughput for small packets is lower than the line rate.

For TCP connections the performance will not be measured as throughput of packets or payloads but as goodput. The goodput is the payload per time that was reliably transmitted through TCP. It depends not only on the raw packet throughput but also on retransmission due to packet drops and waiting time for ACKs.

The graphs will show the mean value of the measurements. The span from minima to maxima is displayed as black line.

6.1 usnetd on netmap

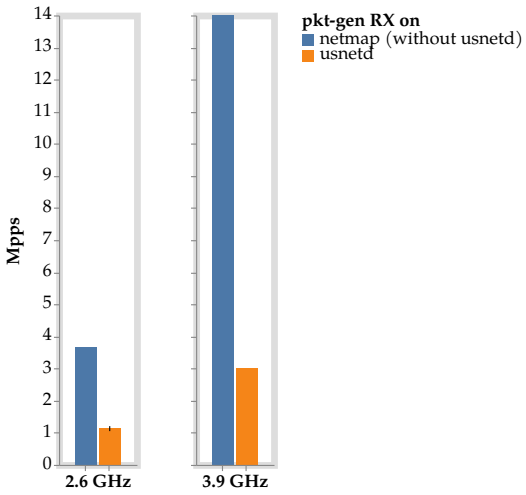

Figure 6.1: Packet matching speed of usnetd compared to directly using netmap with a single thread

The packet matching speed of usnetd was measured with the pkt-gen tool for netmap. Two equal machines were set up with a direct connection of 10G NICs. On both machines the pkt-gen tool used a static netmap pipe to connect to the usnetd switch. The switch uses netmap's zero-copy forwarding. On the sending machine pkt-gen generated UDP traffic which usnetd forwarded to the NIC. On the receiving machine a usnetd rule matched these UDP packets and forwarded them to a pkt-gen instance to measure the Mpps value. The experiment was done for two directly connected 2.6 GHz machines and for two directly connected 3.9 GHz machines. In addition, as baseline test without packet matching and forwarding, pkt-gen was also used without usnetd for both sender and receiver to see the maximal number of packets per second that can be received from netmap with single-thread usage.

Figure 6.1 shows how many minimal-sized UDP packets pkt-gen received per second. On the 3.9 GHz machines pkt-gen received the maximal 14 Mpps on a single core when netmap is directly used. With usnetd for NIC access on the 3.9 GHz machines pkt-gen only receives 3 Mpps. On the 2.6 GHz machines pkt-gen directly on netmap does not receive the maximal 14 Mpps but only 3.6 Mpps with a single core. With usnetd for NIC access pkt-gen just reaches a rate of 1.1 Mpps.

To demonstrate the impact of receiving less than 14 Mpps I measured the UDP packet throughput of pkt-gen on the 2.6 GHz machine for various packet sizes. As measured previously usnetd and single-thread direct usage of netmap both just receive 1.1 and 3.6 Mpps on the 2.6 GHz machines. Therefore, a throughput less than the line rate is expected for small packets. Figure 6.2 shows the throughput in MBit/s as observed by the receiving pkt-gen instance. The varying packet size for the IP packet includes a 14 byte Ethernet header. For maximal-sized packets pkt-gen on both usnetd and netmap received at line rate. Already for 500 byte packets pkt-gen on usnetd received less than half of the line rate while pkt-gen directly on netmap was still close to the line rate. For minimal-sized packets both just received at a fraction of the line rate.

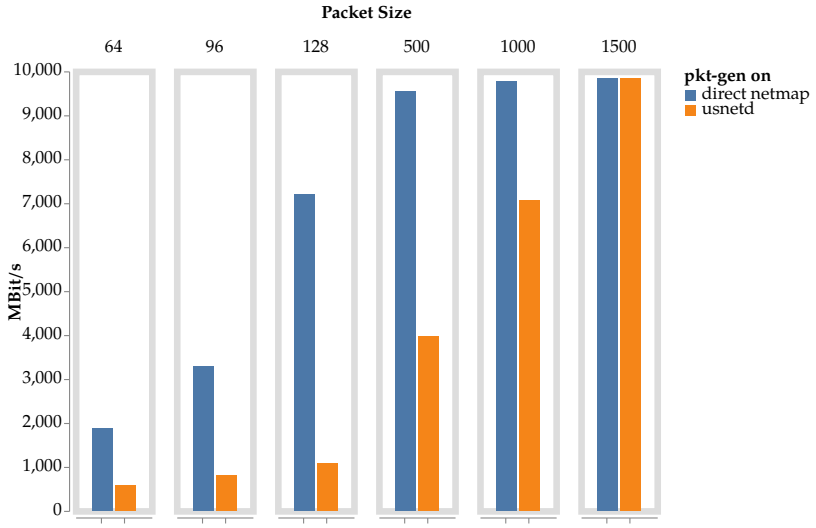

Figure 6.2: Throughput of usnetd compared to directly using netmap (only 2.6 GHz)

6.2 usnet_sockets with smoltcp and netmap

The performance of the userspace networking library usnet_sockets was evaluated by measuring the TCP goodput for a single stream, and the number of completed short-lived connections in parallel per second. The library's socket types are modeled around smoltcp and, therefore, were compared to directly using smoltcp's non-blocking sockets with waiting in a main loop. The strict single-thread socket types of the usnet_sockets library without a background thread introduce only blocking IO above smoltcp. The multi-thread capable socket types with a background thread, however, also add a global mutex lock and the need for context switches. Both variants of the library were compared with smoltcp to see the introduced overhead of blocking IO separately from the overhead of locking and context switches.

In the following experiments I will denote the three socket APIs as "smoltcp"for the direct usage of smoltcp sockets, "usnet_sockets (no BGT)"for single-thread socket types of the usnet_sockets library without a background thread, and "usnet_sockets"for the multi-thread capable usnet_sockets with a background thread and locking. Only the zero-copy socket calls were used. The socket buffers have a size of 500 kB for receive window scaling which suffices to saturate a directly connected 10G link between the test machines. Each process was pinned to a core. The background thread and the application thread were pinned to the same core unless stated otherwise. Due to the global lock both cannot work at the same time. Pinning them on one core reduced the impact of the process scheduler.

Packet IO in smoltcp consumes and emits as many packets as possible before continuing with application logic and waiting. Each fetched packet is immediately processed in the network stack and if applicable a response packet is send out. There are no delayed ACKs in smoltcp and, therefore, most incoming packets will lead to an immediate answer packet. Since there is no congestion control,

packet drop is likely to occur which induces retransmission and hurts the goodput. There are also no selective ACKs which means that even successfully transmitted packets can be send out multiple times. Therefore, one cannot expect the TCP goodput to match the UDP throughput of pkt-gen.

The first experiment measured the TCP goodput of the combined solution of usnet_sockets as socket API and usnetd as switch for shared NIC access. Netmap pipes were used as IPC channels for packets. The switch used zero-copy forwarding and accessed the NIC with netmap. Implicit TX with netmap was used in the switch as well. The two 2.6 GHz machines were directly connected with 10G NICs. One machine acted as TCP sender, one as receiver, whereas receiver and sender always use the same socket API. The two 3.9 GHz machines have the same experiment setup and the measurement was repeated. They have a faster single-core performance than the slower 2.6 GHz machines where the netmap single-core usage was degraded. Since the TCP packets of the sender mostly use the maximal Ethernet frame size, a raw throughput at line rate similar to the UDP pkt-gen measurement is expected. But packet drop can happen both at the switch and the netmap pipe. If one part is too slow to receive all packets, the goodput is expected to go down due to retransmission.

Figure 6.3 shows the TCP goodputs for the 2.6 GHz and the 3.9 GHz setup. For the 2.6 GHz setup the graph shows a goodput around 2 GBit/s. For the 3.9 GHz machines the graph shows a goodput of 9.43 GBit/s which is quite close to the line rate. The results for the 3.9 GHz setup are competitive with Linux which can also saturate a 10G line rate due to its multicore architecture. 40G hardware would be needed to find out the maximal single-thread TCP performance of usnet_sockets on usnetd. To find the bottleneck in the 2.6 GHz setup which prevents a line rate goodput further experiments are needed.

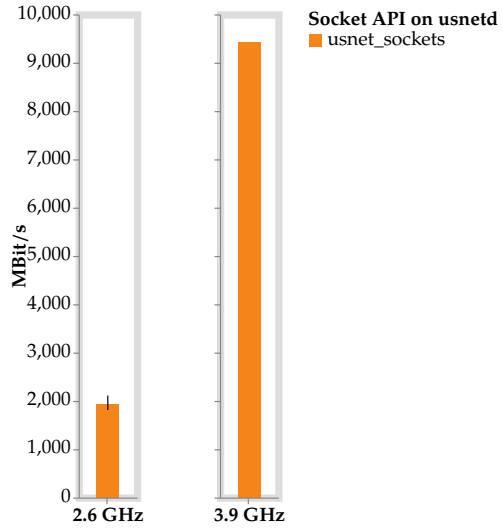

Figure 6.3: TCP goodput with usnetd on netmap for shared NIC access, using netmap pipes as IPC channel for packets

The second experiment used only the 2.6 GHz setup but varied the configurations to find the bottleneck. To measure the synchronization and context switch overhead in the multithread-capable

usnet_sockets API it is compared to single-thread “usnet_sockets (no BGT)”API without a background thread and locking, as well as to directly using the smoltcp API. But since usnetd with the slower netmap performance and packet matching could also be the bottleneck, the whole experiment is repeated without usnetd by using directly netmap for NIC access within the socket APIs. This saves packet matching and the forwarding through netmap pipes but gives higher load to the thread where the socket API does the netmap syscall. The netmap syscalls do not copy packets and work on a batch of packets as when the switch is used. Also implicit TX was used as in the switch, so that packet are sent out as batch in the same syscall that waits and receives the new batch. Figure 6.4 shows on the left side the TCP goodput measurements for the three socket APIs when usnetd is used. The goodput of “smoltcp”varies between 3 and 4 GBit/s, “usnet_sockets (no BGT)”varies between 2.3 and 3 GBit/s, and for “usnet_sockets”is around 2 GBit/s. We can see that there is some overhead both due to the multithread synchronization and the blocking socket logic becaues directly using smoltcp was the faster than “usnet_sockets (no BGT)”and this was already faster than “usnet_sockets”. But still the performance is not even half the line rate for one of them. On the right Figure 6.4 shows the TCP goodputs of the socket APIs without usnetd by directly using netmap. TCP goodput on netmap for “smoltcp”is 6.5 GBit/s, “usnet_sockets (no BGT)”varies between 3 and 5.5 GBit/s, and for “usnet_sockets”is slightly more than 2 GBit/s. All goodputs are much below the line-rate but “usnet_sockets (no BGT)”and “smoltcp”doubled their goodput when usnetd is taken out of the path. For “usnet_sockets”however there was almost no difference whether usnetd is used or directly netmap.

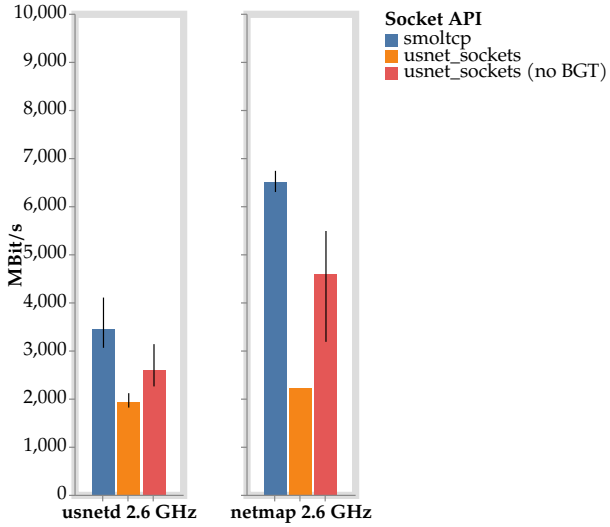

Figure 6.4: TCP goodput on the left with usnetd for shared NIC access using netmap pipes as IPC channel for packets, on the right with netmap for direct NIC access

The following two experiments test the performance when the socket APIs do not use the netmap interface. The possible smoltcp interface options for NIC access in usnet_sockets besides netmap interfaces are Unix domain sockets as IPC channel to usnetd, and directly using macvtap interfaces. For both each packet transfer needs a syscall which copies the packet to either the switch or the NIC interface.

The third experiment uses the three socket APIs on usnetd but now with Unix domain sockets as IPC channels for packets instead of netmap pipes. NIC access in usnetd via netmap remained. The tests used only the 2.6 GHz machines. The Unix domain sockets use one syscall per packet and have to copy the packet between usnetd to the kernel as well as between the kernel and smoltcp interface. Therefore, the internal difference of the APIs is expected to be less significant compared to the overhead per packet. Figure 6.5 shows almost the same TCP goodput for all socket APIs. For "usnet_sockets"the goodput increased a bit compared to netmap pipes as IPC channel.

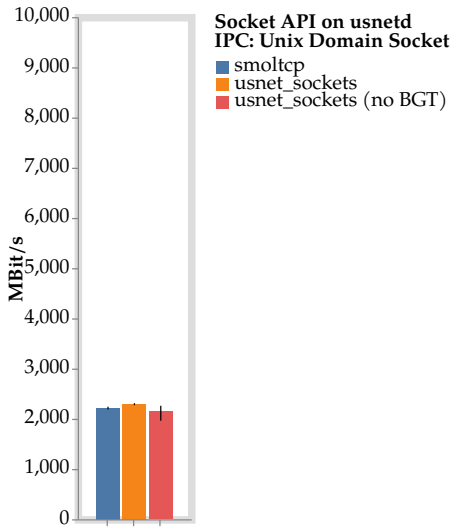

Figure 6.5: TCP goodput with usnetd on netmap for shared NIC access, using Unix domain sockets as IPC channel for packets

The fourth experiment used macvtap in passthru mode for direct NIC access without usnetd. The macvtap syscall receives or transmits only one packet each time and needs to copy it. The two 2.6 GHz machines were directly connected with 10G NICs again. The TCP goodput was measured for the three socket APIs. Figure 6.6 shows that on macvtap all socket APIs reach the same performance of 2 GBit/s. Similar to using Unix domain sockets, the internal differences of the APIs are not visible due to the per-packet overhead.

The fifth and last experiment compared the Linux kernel network stack to the usnet_sockets library in a HTTP server setting. Each short HTTP request needed to be parsed before it was answered with a short response. The listening socket of the server had an own thread and all new incoming connections spawned own threads. The client had a maximum of 32 TCP connections in parallel. It was tested how many of these short-lived connections the network stack processed per second. This is determined by the speed of the connection setup and the scalability of multiple application threads that access the network stack. As first test environment the two 2.6 GHz machines were connected with 10G NICs, and as second test environment the two 3.9 GHz machines were directly connected. The client was the Apache benchmark tool ab on the Linux kernel network stack. Any hardware offloading was disabled for the kernel network stack because the userspace network stacks

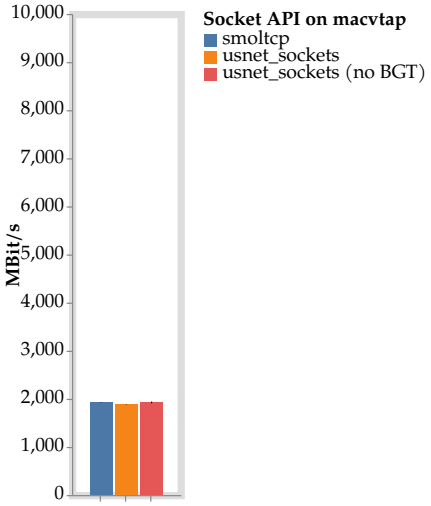

Figure 6.6: TCP goodput with macvtap passthru mode for direct NIC access

do not use it. Multiple cores were still available for the kernel even though smoltcp only operates on a single thread. The server application was compiled with socket types of the Rust standard library to use the kernel network stack. The completed HTTP requests per second as observed by the ab client were measured. The same server application was again compiled with the multithread-capable sockets of the usnet_sockets library that use a background thread. For NIC access the usnetd switch on netmap was used. The IPC channel for packets was a netmap pipe. The background thread was pinned to a core but all application threads were scheduled freely. Again the completed HTTP requests per second as observed by the ab client were measured. The Linux kernel network stack was able to use all cores to process the TCP connections. The application threads could read and write in parallel from the Linux sockets. For usnet_sockets the application threads and the background thread all had no parallel access to the sockets. Figure 6.7 shows the mean of completed

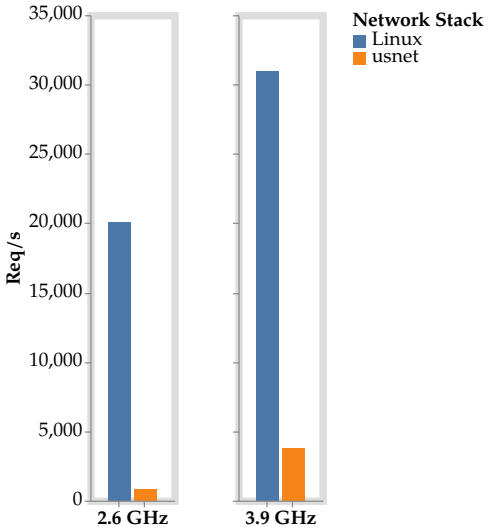

Figure 6.7: HTTP request completion per second for a server using the kernel network stack compared to the same server using usnet_sockets on usnetd (netmap pipes for IPC, netmap for NIC)

requests per second for the Linux network stack and userspace networking on the 2.6 GHz and 3.9 GHz machines. The server using the kernel network stack completed around 20000 requests per second on the setup with the 2.6 GHz machines, and more than 30000 requests on the setup with the 3.9 GHz machines. The server using usnet_sockets with usnetd, however, only completed 850 requests per second on the setup with the 2.6 GHz machines, and 3750 requests per second on the setup with the 3.9 GHz machines.

The previous experiment did not evaluate the scalability of multiple longer TCP streams. This would not only depend on the scheduling of the application threads as for short requests but also on the congestion control algorithm. Since there is no congestion control in smoltcp, such an experiment results mostly not in a fair share of the line rate and the goodputs vary too much for a meaningful evaluation.

6.3 Required Source Code Changes

To actually use memory-safe networking through the usnet_sockets library I argued for a source code approach in Section 4.2. Redirecting the libc functions or syscalls at runtime for any binary would be the other approach. This section evaluates which changes to the code base are needed for a Rust project to use memory-safe userspace networking with usnet_sockets.

As example I chose the HTTP library tiny-http [149] and the web framework rouille which uses tiny-http [150]. A concrete implementation of a web service could use rouille and, thus, indirectly uses tiny-http. When such a web service is installed by building it from source with Rust's `cargo install` command, then ideally a build feature flag should be enough to enable memory-safe userspace networking. The following steps describe which changes were needed to achieve this.

Since tiny-http directly uses the Rust standard library socket types I modified the import statements to use the compatible socket types in usnet_socket. These modifications should not be the default for tiny-http and, therefore, were guarded by a feature flag called "`usnet`". Now the library either uses the kernel network stack or the userspace network stack depending on the build feature flag. Listing 6.1 lists these changes. When porting the library I first needed to add a feature to usnet_sockets which I first saw as low priority. I needed to add support for cloning TCP streams as well as closing the TX or RX half of the connection. In smoltcp, however, the RX half alone cannot be closed without the TX half since it is impossible with TCP to forbid the sender to send more. I decided to ignore such a close action for the RX half. A solution would be to keep a separate RX-closed state in the `TcpStream` socket type of usnet_sockets to forbid future read calls.

```
diff --git a/Cargo.toml b/Cargo.toml
index aaf8eb1..65720d7 100644
--- a/Cargo.toml
+++ b/Cargo.toml
 -12,6 +12,7  repository = "https://github.com/tiny-http/tiny-http"
 [features]
 default = []
+usnet = ["usnet_sockets"]

[dependencies]
+usnet_sockets = { git = "https://…/usnet_sockets", optional = true }

diff --git a/src/lib.rs b/src/lib.rs
index a768c80..e457205 100644
--- a/src/lib.rs
+++ b/src/lib.rs
 -119,6 +119,9  extern crate chrono;

+#[cfg(feature = "usnet")]
+extern crate usnet_sockets;

-use std::net::{ToSocketAddrs, TcpStream, Shutdown};
+#[cfg(feature = "usnet")]
+use usnet_sockets::{TcpStream, TcpListener};
+#[cfg(not(feature = "usnet"))]
+use std::net::{TcpListener, TcpStream};
+use std::net::{ToSocketAddrs, Shutdown};

 -243,7 +250,7  impl Server {

-            let listener = try!(net::TcpListener::bind(config.addr));
+            let listener = try!(TcpListener::bind(config.addr));

diff --git a/src/util/refined_tcp_stream.rs b/src/util/refined_tcp_stream.rs
index 474afbb..5af36f7 100644
--- a/src/util/refined_tcp_stream.rs
+++ b/src/util/refined_tcp_stream.rs
 -14,7 +14,11

-use std::net::{SocketAddr, TcpStream, Shutdown};
+#[cfg(feature = "usnet")]
+use usnet_sockets::TcpStream;
+#[cfg(not(feature = "usnet"))]
+use std::net::TcpStream;
+use std::net::{SocketAddr, Shutdown};
```

Listing 6.1: tiny-http: Patch for usnet_sockets as build feature

There are no modifications needed in the web framework rouille except for a special reverse proxy feature. Porting it is currently not possible because it requires support to set the connection timeout. Instead of removing the timeout configuration depending on the build flag or adding the timeout calls to usnet_sockets, I decided to leave this part untouched and use the kernel sockets. If a Rust web service does not use these reverse proxy connection, the developer could already combine the unmodified rouille with the new usnet feature of tiny-http and have full memory-safe TCP/IP.

If it would use the reverse proxy connection, then the kernel network stack is used for outgoing connections. As long as the connection goes to the internal network, as common for a reverse proxy setup, no exposure of the kernel network stack to untrusted packets takes place.

A clean way for userspace networking support in any Rust project that uses the rouille web framework is adding a new usnet build feature to enable the respective tiny-http build feature. This breaks down to adding the line `usnet = ["tiny_http/usnet"]` in the `[feature]` section of the `Cargo.toml` build system file. Additionally, if the tiny-http patch is not included in the upstream repository, a path to the patched tiny-http library is needed in the `[patch.crates-io]` section.

Another option is to modify the rouille web framework `Cargo.toml` file in the same way to expose the build feature. This does not make a big difference for the developer of the Rust web service, since still an entry in `Cargo.toml` would be required, this time `usnet = ["rouille/usnet"]` which also specifies that the usnet feature will be passed through to the library.

In conclusion source code changes were only necessary in the library where the socket types are imported. Projects which depend on this library, even indirectly, just need to add a build flag for memory-safe userspace networking with usnet_sockets.

7 Discussion

This chapter will discuss the results of the previous section and assess the state of the prototypes. It will also propose alternative implementation choices and have a overall outlook of benefits and limitations.

7.1 Experimental Results

The first experiment revealed for the baseline test a degraded performance for netmap on the 2.6 GHz setup with single-core usage and small packets. On the 3.9 GHz setup, however, single-core netmap could use the full link speed with any packet size. The experiment for usnetd on both setups showed that usnetd could not forward small packets at line rate to a netmap pipe. Since usnetd like pkt-gen directly on netmap also uses a single-core, usnetd inherits the degraded performance on the 2.6 GHz setup. The difference between pkt-gen directly on netmap and pkt-gen as endpoint on the usnetd switch is that usnetd has to match on all packets. It also uses one big syscall to receive packets from the NIC and the control socket, and to send out packets on the netmap pipe. Performance was not the main goal of the usnetd design and packet parsing and matching also comes at a cost. That said, a multithread architecture would have the biggest benefits for the 2.6 GHz machine because the single-thread netmap performance is not optimal. In addition, either a faster hash table could be used or the last matched rule cached to save the lookups into the hash table. Before any of these optimizations would be done, other features such as forwarding of packets with IP fragmentation and additional matches for broadcast and multicast packets should first be implemented. A good target goal for the performance on the 2.6 GHz machines would be that introducing usnetd has no negative impact. This is the case later for the 3.9 GHz machines in the throughput experiment where usnetd did not hurt the TCP performance compared to direct NIC access with netmap.

The first experiment for usnet_sockets on usnetd successfully demonstrated a 10G line-rate TCP goodput for the 3.9 GHz setup. However, the slower 2.6 GHz setup did not perform well and no line-rate TCP goodput was possible. Since line-rate throughput with UDP on pkt-gen was demonstrated on the slower system and one could have expected TCP to be in a similar range despite the more involved calculations. As smoltcp lacks some TCP features to make it more robust against packet drop, packet drop hurts the goodput more significantly because of large retransmissions. One

reason for packet drop can be that the single-thread netmap performance already had problems to handle many packets. Packet acknowledgment in smoltcp sends out more packets than necessary which increases the work for the netmap NIC syscall compared to only receiving the maximal-sized UDP packets in pkt-gen. The second reason also linked to the single-thread performance is the computation time needed for TCP because it increases the time between two netmap NIC syscalls, leading to possible packet drop. Parallel processing of packets in smoltcp could scale the network stack on multiple cores to reduce the computation time. But before these factors of using smoltcp are considered, the introduced overhead of the usnet_sockets API as well as using usnetd needs to be examined. The second experiment gave more insights on the overheads which contributed to the bad performance and where optimizations are needed to improve it on older systems without changing smoltcp.

The second experiment used only the 2.6 GHz setup. It compared the three socket APIs to see the overhead of multithread synchronization in usnet_sockets compared to using a single-thread API without a background thread and compared again to using directly smoltcp without any blocking logic in the socket calls. First usnetd was used for NIC access connected through a netmap pipe as IPC channel for packets. In contrast to a UDP stream for a TCP stream the packet matching load for usnetd is higher because due to the duplex nature of TCP usnetd not only has to parse and match all incoming packets but also all outgoing packets each time and then forward all at once. On the 2.6 GHz machine only 1.1 Mpps could be matched with usnetd, while 0.8 Mpps would be needed just for matching the incoming packets at line rate, but I expect that almost the same amount of packets has also to be matched the same time on the transmission path because the ACKs generated in smoltcp are not packed together as delayed ACKs.

The relation of the three socket API variants behaved as expected with increasing overheads. The single-thread “usnet_sockets (no BGT)”API was sometimes much slower than directly using the smoltcp API whereas both do just use a single thread. The socket calls do not only have more function calls but also a different logic for waiting and packet RX/TX. Some optimizations may be possible in regards to function inlining and finding cases where waiting could be avoided. The multi-thread capable usnet_sockets API with a background thread had the slowest performance because it needs locks and notifications to coordinate the access of the application thread and background thread to the socket set in smoltcp. The logic for waiting and packet RX/TX differs much from the other socket APIs. Its overhead between the threads can on one hand probably cause a higher delay due to context switches before a response packet is sent out. On the other hand it is also likely that waking up the background thread to send out packets causes in total more netmap syscalls then in the other socket APIs, leading to a smaller batch size. Another issue is that for any processed packet which potentially changed the socket state, the waiting application thread is unblocked and gets the global socket lock to test its waiting condition. A more fine grained read-write locking on single sockets, as well as a notification mechanism that tests the conditions beforehand would be necessary improvements.

These results when usnetd is used were compared to directly accessing the NIC with netmap out of the socket APIs. The second part of the experiment indeed showed without usnetd the performance for the “usnet_sockets (no BGT)”API and directly using the smoltcp API doubled. This means that usnetd was the bottleneck for these configurations and caused packet drops. However, the multi-thread capable usnet_sockets API with a background thread did not show a higher performance and, thus, itself was the bottleneck due to its locking and notification overhead.

The third experiment used the socket APIs with usnetd on netmap again, but connected to it though Unix domain sockets as IPC channel for packet exchange between both. This means that each packet is directly forwarded from the NIC to the endpoint with a syscall in usnetd. Also always a single packet is fetched or sent out by the smoltcp interface. This led to the same performance of all socket APIs because it hid the architectural differences on how waiting and notification works. For usnet_sockets with a background thread this even improved performance slightly which can be explained by a better scheduling behavior of feeding the endpoint with a continuous flow instead of packet batches.

The fourth experiment for the socket APIs used macvtap for direct NIC access. Using macvtap in passthru mode or L2 bridge mode maybe seems interesting for 1G or 2.5G NICs but is too limiting on 10G NICs. The main advantage is that no kernel modules and special drivers are needed and macvtap’s interface was simpler to program than netmap support. With more research on macvtap’s multiqueue functionality and optimal usage, some performance gains seem possible. But extra copy is likely to stay because the kernel only has zero-copy support for in its vhost switch for paravirtualized virtio NICs.

The last experiment evaluated the performance for handling many short TCP connections with separate application threads. Measured was the completion of HTTP requests per second for usnetd on netmap, with a netmap pipe as IPC channel to the multi-thread capable usnet_sockets API with a background thread. As comparison the same server program was tested using the Linux network stack. This is the only experiment where the sender keeps the same configuration because the Apache benchmark tool does not use the Rust network stack. Therefore, this experiment only tells something about the speed of accepting connections in the server from a main application thread and moving them to a separate thread, not about creating sockets in already separate threads in the client. The higher single-thread performance of the 3.9 GHz system had an improvement by factor 4.3 for the completion of requests per second compared to the 2.6 GHz system. The improvement is not linear with factor 1.5 because the 2.6 GHz system has a degraded netmap performance which usnetd inherits. Still improving only the packet matching rate in usnetd would not give enough improvement to have a performance equal to the Linux network stack. The usnet_sockets locking and notification overhead is the main bottleneck because the background thread wakes up every application thread to check its waiting conditions. Fine-grained locking and notification mecha-

nisms can improve this as mentioned, later on also a parallel packet processing in smoltcp should be investigated.

Section 6.3 showed the necessary source code changes for applications to use usnet_sockets with a build flag. For a project such as the HTTP library which directly imports the socket types it is best to incorporate the import statement changes into the official source code tree. Maintaining a patched version is possible but can quickly get outdated. For a project that depends on a ported library only the build metadata needed changes. Such a small patch can easily be maintained out of the tree, or manually applied as needed. Not all parts of a project have to be ported as demonstrated with the web framework. Userspace networking was applied for the server sockets through the ported HTTP library. This partial application enables to focus on the incoming connections when porting an application. The reverse proxy functionality, however, would use a kernel socket for the outgoing connection. Deciding if using the kernel network stack is critical depends on whether this reverse proxy feature is actually used, and if it connects to a process at localhost, a trusted other IP, or whether an attacker can supply an own IP to connect to and then send malicious TCP packets.

Porting the reverse proxy feature to usnet_sockets needs support for connection timeouts in the `TcpStream` socket types of usnet_sockets. This is just one case where usnet_sockets currently lacks important features.

Many networking libraries use the socket wrapper types of the Tokio event loop for asynchronous IO. Tokio uses the mio library for non-blocking socket types. Currently mio cannot be ported because usnet_sockets currently has neither non-blocking calls nor a wrapper for the `epoll` syscalls in Linux to poll the readiness of usnet_sockets sockets. If mio is ported many Rust projects would directly benefit. Unfortunately this could not be demonstrated with the current state of the prototype.

The exact behavior of the Rust standard library and the kernel sockets is also not easy to model. Even if all features are implemented I expect some corner cases where both implementations behave differently. The Rust standard library even notes platform-specific behavior, such as different return values for Linux and macOS when calling the TCP `shutdown` function a second time. Rust applications written for Linux may need workarounds until usnet_sockets simulates the exact same behavior as the Rust standard library on Linux sockets. Through build flags it is possible to enable these workarounds only for userspace networking.

7.2 Benefits and Weaknesses of usnetd and usnet_sockets for Memory-safe Network Services

With usnet_sockets an existing simple code base for a Rust networking service can benefit from memory-safe TCP/IP in userspace. It can be deployed with an own IP on a L2 switched or dedicated NIC with macvtap or netmap/VALE. With usnetd multiple instances can share the same IP with a firewalled host kernel network stack. For a system with high single-core performance the TCP goodput was close to the line rate in a lab setting with 10G NICs.

The main weaknesses for real world deployments at the current state in smoltcp are lack of congestion control, IP fragmentation, path MTU discovery and probing of zero windows. Optionally the TCP performance could be improved with selective and delayed ACKs, avoiding the silly window syndrome, and using Nagle's algorithm. The main weakness of the prototype implementation of usnet_sockets is lacking feature parity with the Rust standard library socket types and no support for the mio and Tokio libraries. Optionally the performance could be improved with multithreading within smoltcp, multiple background threads for NIC access, fine-grained locking of the sockets, and optimized unblocking of application threads. The main weakness of usnetd is the missing support for broadcast, multicast, and IP fragmentation. For greater compatibility it should also support other backends for NIC access, namely macvtap, AF_XDP, and DPDK. Optionally the performance could be improved with parallel packet matching and multiple NIC queues.

The prototypes of usnetd and usnet_sockets compare to the existing solutions as follows. The memory-safe network stacks mirage-tcpip, HaNS, and smoltcp either run in unikernel/microkernel VMs or as process on TAP devices. None of them integrates with the kernel loopback interface for internal communication. Since they use TAP devices, they need L2 switching and separate external IPs. It would have been possible to port one of them to MultiStack, TAPM, or swarm to share an IP. The VALE module MultiStack and swarm, however, both do not firewall the kernel nor do use a memory-safe language for packet matching. TAPM relies on hardware for the packet matching but also does not firewall the kernel. They are not suitable to deploy memory-safe networking services. In addition TAPM is not even published, MultiStack unmaintained, and for swarm the kernel needs an own MAC and IP. In contrast to the previous software switches for IP sharing, usnetd is suitable to deploy memory-safe network services because it firewalls the kernel, and itself also uses memory-safe packet matching in software without relying on NIC hardware features. It also provides both a dynamic creation of switch endpoints and port registration as well as a static configuration. Table 7.1 lists the switch solutions for userspace network stacks and whether they allow to share the IP with the kernel, have memory-safe packet matching, firewall the kernel, allows the endpoints to use different IPs and still connect them with each other, and has support for dynamic creation of endpoints. Connecting endpoints with different IPs is required if one network stack cannot share the IP but still needs to be connected with other network stacks on the switch.

For TAPM this needs an external hardware switch in VEPA hairpin mode because packet matching is done in the NIC only for incoming packets. For swarm only one IP can be shared but swarm can be attached to VALE to reach the kernel network stack on a different IP. Dynamic creation of endpoints with VALE or MultiStack needs an additional way to manage which endpoint names are still free.

Table 7.1: Software Switches multiplexing the NIC for Host Kernel and Memory-safe Network Services

Switch	IP Sharing	Memory-safe TCP/IP Parsing	Firewalls Kernel	Connects Endpoints with different IPs	Dynamic Creation
Linux macvtap/L2 bridge	no	-	no	not host kernel	yes
VALE L2 switch	no	-	no	yes	needs code against name clashes
TAPM	yes	in hardware	no	needs VEPA hairpin	?
swarm	not kernel	no	no	needs VALE	yes
VALE-MultiStack	yes	no	no	yes	needs code against name clashes
usnetd	yes	yes	yes	yes	yes
VALE-bpf + global eBPF bytecode	yes	yes	yes	yes	needs code against name clashes
AF_XDP + eBPF code per queue	yes	yes	userspace helper as monitor	needs VEPA hairpin	needs attaching via shared region
PFQ + pfq-lang per endpoint	yes	yes	userspace helper as monitor	needs VEPA hairpin	yes

The second part of Table 7.1 includes not complete solutions but programmable switching solutions relying on in-kernel bytecode execution. They are suitable for deployment of memory-safe network services if a suitable switching logic is programmed with similar forwarding rules as usnetd. A bytecode for VALE-bpf could come close to what usnetd achieves in userspace. Bytecodes for AF_XDP or PFQ would need userspace helpers to monitor outgoing connections of the kernel and update the forwarding rules. With AF_XDP or PFQ the endpoints are not connected with each other in the software switch. A VEPA hairpin mode hardware switch would be needed to connect them. Seeing outgoing packets of the kernel network stack for the firewall rules needs an additional monitor process. Since AF_XDP is now included in the Linux kernel it is the most interesting target. Supporting dynamic management of multiple endpoints, however, is more challenging compared to PFQ and VALE-bpf. It involves queue management, attaching to a shared region of one of multiple queues and updating the matching rules for the eBPF bytecode.

Both usnetd and usnet_sockets together provide a full solution for deploying multiple memory-safe network services in Rust as Linux processes. The only userspace TCP/IP socket API for Rust until now was provided by smotcp sockets which is not compatible with the socket types of the standard library and requires a different application architecture. By choosing Rust for usnet_sockets the networking in Rust can expand their memory-safety easily to TCP/IP through userspace networking.

Table 7.2: Solutions for Memory-safe TCP/IP

Solution	Architecture	Implementation	In-app TCP/IP	Service Language	Note
MirageOS	Unikernel VM	OCaml	yes	OCaml	uses mirage-tcpip
mirage-tcpip	Process on bridged TAP	OCaml	yes	OCaml	
HaLVM	Unikernel VM	Haskell	yes	Haskell	uses HaNS
HaNS	Process on bridged TAP	Haskell	yes	Haskell	
gVisor	Process in Linux-compatible paravirtualized container	Go	no	any	memory safety only with data race detector
RedoxOS	Microkernel OS VM	Rust	no	any if ported	uses smoltcp
smoltcp	Process on TAP	Rust	yes	Rust	
usnet_sockets	Process on macvtap/netmap/usnetd	Rust	yes	Rust	uses smoltcp

It is not required to use Rust and usnet_sockets on usnetd. Porting other userspace network stacks, such as mirage-tcpip or HaNS, should be relatively simple because TAP devices and Unix domain sockets for usnetd IPC both use simple syscalls. The network stack of the Linux-compatible paravirtualized container runtime gVisor could be ported as well. However, the network stack is only memory-safe if gVisor is compiled with the Go race detector. Any Linux program running within this container will directly use the Go network stack when it creates kernel sockets.

Unikernel VMs with netmap passthru support, such as MirageOS on solo5-netmap, are already suitable as endpoints on usnetd. However, they currently need a static configuration of open ports. For dynamic creation of endpoints the hypervisor monitor needs to be ported to request the netmap pipe from usnetd and register ports for listening sockets. VMs without netmap passthru support, such as HaLVM unikernels and RedoxOS VMs currently cannot be endpoints on usnetd because the hypervisor works with TAP devices as backend for the virtual NIC.

Table 7.2 lists the solutions for memory-safe network stacks, whether they run as process, microkernel VM, or unikernel VM, which programming language they use for the network stack, and which programming languages they support. RedoxOS and gVisor allow for the service to use any programming language because the network stack is provided by the OS. How the performance of userspace networking with usnet_sockets compares to the other solutions was not explored in this thesis.

7.3 TCB and Limitations of Memory-safe Networking

The presented solution addresses the requirements of the analysis in Section 3.1 for deploying memory-safe networking on Linux alongside the kernel network stack. The requirements were based on a thread model which assumes that the Linux kernel TCP/IP implementation will have remote code execution vulnerabilities. It assumes that the attacker controls the IP headers but cannot spoof IPs. Further, L2 handling and network drivers form the TCB. In fact, for many servers the

main attack surface is in the application layer due to memory errors in C/C++, but memory-safe applications do not suffer from this. For a Rust network service with usnet_sockets for memory-safe TCP/IP in userspace the TCB consists of the following parts where unsafe code is involved. The Rust standard library internally uses unsafe Rust for some types it exposes, or for syscalls to the kernel. There are some library dependencies for usnet_sockets and smoltcp which do also include unsafe Rust. The major unsafe dependencies are `byteorder`, `log`, `netmap_sys`, `rand`, `parking_lot` for mutex and condvar, and `nix` for file descriptor handover with Unix domain sockets and other syscall wrappers. In usnetd and usnet_sockets itself have short unsafe Rust code for the netmap data structures and syscalls, macvtap syscalls, and file descriptor handover. My code for netmap has the largest part of unsafe Rust and is the most critical because the netmap data structures rely on setting the correct buffer pointers and there was no peer review. In my initial implementation of the zero-copy forwarding the the assumption of L2 as TCB failed. One variable name was wrong and resulted in multiple concurrent usages of the same netmap buffers. This caused memory corruption which triggered a Rust bounds check assertion. The possibility of abusing this memory corruption for remote code execution is rather low but still exists.

Using a memory-safe language for a network stack does not prevent DoS attacks. If a malicious packet causes a long-running computation or a runtime panic this can render the service unavailable. Also the implementation of a protocol is not verified to be correct and reliable. Even for valid packets it can have runtime panics, close a TCP connection unexpectedly, or do not proceed with communication.

Memory-safe programming languages do also not protect against information leaks through side-channel attacks. Usually side-channels leak the programs own state, e.g., to infer a private cryptographic key. Side-channel attacks based on processor flaws, however, can leak the micro-architectural state of the CPU. This state can depend on the content of memory areas the program even did not use but were accessed due to speculative execution. NetSpectre demonstrated leaking memory remotely through latency measurements of packets .

Hardware flaws can also cause memory-corruption for any programming language. The rowhammer effect is common for DDR3 and DDR4 RAM and causes bit flips in neighboring cells if a memory cell is accessed in a certain pattern. Nethammer and Throwhammer demonstrated corrupting memory remotely by sending a crafted flow of packets [151, 152].

8 Conclusion

This work argued that network services should be implemented in memory-safe languages. They prevent remote code execution vulnerabilities due to memory corruption. For programs on a general operating system such as Linux the network stack is still written in C. Multiple memory-corruption bugs with high security impact are found every year. Mitigations, software testing, and fault isolation are insufficient approaches. Unfortunately software verification is currently not ready to ensure memory safety for the Linux kernel.

Network stacks in memory-safe languages were introduced in the background section. They address the problem but in practice are mostly used in non-Linux VMs and require L2 switching. The background section introduced userspace networking which is mainly used for performance reasons. Besides performance it also enables a processes to use a memory-safe network stack. Multiple network stacks can share a NIC through L2 switching, and there are some solutions to share an IP with L4 switching.

In the analysis chapter I assumed a threat model under which L2 switching and the network driver can form a TCB. If, however, L4 switching is used, it should be memory-safe. Both L2 and but specially L4 switching need a protection mechanism for the host kernel network stack. I argued for a firewall that trusts response packets for outgoing connections.

The analysis chapter also compared memory-safe unikernels, memory-safe operating systems, and processes with memory-safe userspace networking in regards to backwards compatibility and TCB. I concluded that memory-safe userspace networking has many advantages. Yet at the current time, there is no userspace networking library that focuses on providing the same API and behavior as the kernel sockets. To ease the porting of an application to memory-safe networking, I argued for a Rust userspace networking library that integrates well with the kernel network stack and shares one IP via L4 switching. I analyzed the existing switches for userspace networking and found that no solutions suffices and argued for a memory-safe switch. Therefore, I analyzed which building blocks a switch for memory-safe network services can use.

Based on the developed requirements for memory-safe network services on Linux I designed two components. The first component is the userspace switch called usnetd. It should have memory-safe packet matching and firewall the host kernel. It is designed for dynamic endpoints which register their open ports. It also allows static configurations for endpoints which do not support

dynamic configuration. The design is not tied to a specific kernel bypass framework and allows for various NIC backends to exist. The second component is a userspace networking library for Rust called usnet_sockets. It is compatible with the standard library, can interact with other the kernel network stack, and makes use of usnetd to share one IP.

I implemented prototypes for usnetd on netmap and usnet_sockets. I reused the existing Rust network stack smoltcp and fixed bugs in its TCP implementation. I modeled a multithread blocking API around it. Both prototypes currently do not have all planned features.

In the evaluation the performance was assessed in order to see possible optimization points in the internal architecture. A HTTP library that is used in a Rust web framework was ported to userspace networking. This gave an example for the required changes in the source code or build metadata. Using usnetd and usnet_sockets in combination provides an easy path to memory-safe networking with Rust on Linux. While usnetd improves on the current state of switches for userspace network stacks and fills the gap for memory-safe networking, usnet_sockets makes memory-safe userspace networking accessible for Rust network services with small porting efforts.

There are, however, many features to be implemented for general usage. Even though a 10G TCP goodput was demonstrated with two connected fast machines, the multicore scalability is an issue. Memory-safe TCP/IP is a solution for the mentioned threat model but still includes a TCB for L2 access. Also only vulnerabilities related to software memory corruption are prevented, other security issues due to implementation and hardware flaws remain.

The source code is published at `https://github.com/ANLAB-KAIST/usnet_sockets`, `https://github.com/ANLAB-KAIST/usnetd`, and `https://github.com/ANLAB-KAIST/usnet_devices` under a permissive free software license. The scripts to reproduce the measurements are also provided. Future work should address missing features in usnetd, usnet_sockets, and smoltcp to fully implement the design. More NIC backends and userspace network stacks should be ported to usnetd. In particular integration of AF_XDP as a new kernel bypass mechanism in Linux should be tested as replacement for netmap in usnetd. Finally memory-safe TCP/IP could be used with any program through redirecting the libc socket calls at runtime with LD_PRELOAD, or a local memory-safe SOCKS proxy. Performance comparisons between TCP/IP implementations, memory-safe and not, may help to optimize the memory-safe implementations and promote them. Memory safety prevents remote code execution vulnerabilities due to memory corruption. Therefore, expanding memory safety into the network stack improves the security of network services. This work made memory-safe network stacks easily usable on the general purpose operating system Linux.

Bibliography

[1] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. **NetSpectre: Read Arbitrary Memory over Network**, 2018. https://arxiv.org/abs/1807.10535.

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. **A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World**. *Commun. ACM*, 53(2): 66–75, February 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. http://doi.acm.org/10.1145/1646353.1646374.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. **KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs**. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. http://dl.acm.org/citation.cfm?id=1855741.1855756.

[4] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. **SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis**. In *IEEE Symposium on Security and Privacy*, 2016. https://ieeexplore.ieee.org/document/7546500.

[5] Serkan Özkan. **CVEdetails.com**, 2018. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html.

[6] CORE Security Technologies Advisories. **CORE-2007-0219: OpenBSD's IPv6 mbufs remote kernel buffer overflow**, 2007. https://lwn.net/Articles/225947/.

[7] Kevin Backhouse. **Kernel crash caused by out-of-bounds write in Apple's ICMP packet-handling code (CVE-2018-4407)**, 2018. https://lwn.net/Articles/225947/.

[8] Mat Martineau, Matthieu Baerts, Christoph Paasch, and Peter Krystad. **How hard can it be? Adding Multipath TCP to the upstream kernel**. *netdev 2018*, 2018. https://www.netdevconf.org/0x12/session.html?how-hard-can-it-be-adding-multipath-tcp-to-the-upstream-kernel.

[9] Marta Rybczyńska. **Writing network flow dissectors in BPF**, 2018. https://lwn.net/Articles/764200/.

[10] Daniel Borkmann. **net: add bpfilter**, 2018. https://lwn.net/Articles/747504/.

[11] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. **A Readable TCP in the Prolac Protocol Language**. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 3–13, New York, NY, USA, 1999. ACM. ISBN 1-58113-135-6. doi: 10.1145/316188.316200. http://doi.acm.org/10.1145/316188.316200.

[12] Dave Jones. **trinity Linux system call fuzzer**, 2012. https://github.com/kernelslacker/trinity.

[13] Google, Inc. **syzkaller: unsupervised, coverage-guided kernel fuzzer**, 2015. https://github.com/google/syzkaller.

[14] Linus Torvalds. **sparse: semantic checker for C**, 2003. https://sparse.wiki.kernel.org/index.php/Main_Page.

[15] Dan Carpenter. **smatch: static analysis tool for C**, 2005. https://repo.or.cz/w/smatch.git.

[16] Jiří Slabý. *Automatic Bug-finding Techniques for Large Software Projects*. PhD thesis, Masarykova univerzita, Fakulta informatiky, 2014. https://is.muni.cz/th/ehqsd/dis.pdf.

[17] Linux Verification Center. **astraver: Linux Deductive Verification**, 2015. http://linuxtesting.org/astraver.

[18] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and A. V. Khoroshilov. **Configurable Toolset for Static Verification of Operating Systems Kernel Modules**. *Program. Comput. Softw.*, 41(1):49–64, January 2015. ISSN 0361-7688. doi: 10.1134/S0361768815010065. http://dx.doi.org/10.1134/S0361768815010065.

[19] Madanlal Musuvathi and Dawson R. Engler. **Model Checking Large Network Protocol Implementations**. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association. http://dl.acm.org/citation.cfm?id=1251175.1251187.

[20] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. **Conccinelle: Program Matching and Transformation Tool for Systems Code**, 2005. http://coccinelle.lip6.fr/.

[21] Coverity, Inc. **Coverity**, 2002. https://scan.coverity.com/.

[22] Jonathan Corbet. **One year of Coverity work**, 2014. https://lwn.net/Articles/608992/.

[23] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. **Keep Net Working - on a Dependable and Fast Networking Stack**. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. http://dl.acm.org/citation.cfm?id=2354410.2355161.

[24] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. **Network Stack As a Service in the Cloud**. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 65–71, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5569-8. doi: 10.1145/3152434.3152442. http://doi.acm.org/10.1145/3152434.3152442.

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. **Unikernels: Library Operating Systems for the Cloud**. *SIGPLAN Not.*, 48(4):461–472, March 2013. ISSN 0362-1340. doi: 10.1145/2499368.2451167. https://dl.acm.org/citation.cfm?doid=2499368.2451167.

[26] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. **Turning Down the LAMP: Software Specialisation for the Cloud**. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. http://dl.acm.org/citation.cfm?id=1863103.1863114.

[27] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. **Jitsu: Just-in-time Summoning of Unikernels**. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 559–573, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. http://dl.acm.org/citation.cfm?id=2789770.2789809.

[28] Galois, Inc. **The Haskell Lightweight Virtual Machine (HaLVM): GHC running on Xen**, 2017. https://github.com/GaloisInc/HaLVM.

[29] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. **Hyperkernel: Push-Button Verification of an OS Kernel**. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 252–269, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132748. http://doi.acm.org/10.1145/3132747.3132748.

[30] RWTH Aachen University. **libhermit-rs: A Rust-based Unikernel for Cloud and High-Performance Computing**, 2018. https://github.com/hermitcore/libhermit-rs.

[31] Antti Kantee. **Environmental Independence: BSD Kernel TCP/IP in Userspace**. AsiaBSDCon, 2009. https://2009.asiabsdcon.org/papers/abc2009-P5A-paper.pdf.

[32] Kevin Elphinstone, Amirreza Zarrabi, Kent Mcleod, and Gernot Heiser. **A Performance Evaluation of Rump Kernels As a Multi-server OS Building Block on seL4**. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 11:1–11:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5197-3. doi: 10.1145/3124680.3124727. http://doi.acm.org/10.1145/3124680.3124727.

[33] Kent McLeod. **Using Rump kernels to run unmodified NetBSD drivers on seL4**, 2017. https://research.csiro.au/tsblog/using-rump-kernels-to-run-unmodified-netbsd-drivers-on-sel4/.

[34] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. **EbbRT: A Framework for Building Per-application Library Operating Systems**. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 671–688, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. http://dl.acm.org/citation.cfm?id=3026877.3026929.

[35] Richard D. Greenblatt, Thomas F. Knight, John T. Holloway, and David A. Moon. **A LISP Machine**. *SIGIR Forum*, 15(2):137–138, March 1980. ISSN 0163-5840. doi: 10.1145/1013881.802703. http://doi.acm.org/10.1145/1013881.802703.

[36] J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon. **The Symbolics Genera Programming Environment**. *IEEE Software*, 4(6):36–45, Nov 1987. ISSN 0740-7459. doi: 10.1109/MS.1987.232087. https://ieeexplore.ieee.org/document/1695856.

[37] D. A. Moon. **Symbolics Architecture**. *Computer*, 20(1):43–52, Jan 1987. ISSN 0018-9162. doi: 10.1109/MC.1987.1663356. https://ieeexplore.ieee.org/document/1663356.

[38] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. **SPIN–an Extensible Microkernel**

for Application-specific Operating System Services. *SIGOPS Oper. Syst. Rev.*, 29(1):74–77, January 1995. ISSN 0163-5980. doi: 10.1145/202453.202472. http://doi.acm.org/10.1145/202453.202472.

[39] Marc E. Fiuczynski and Brian N. Bershad. **An Extensible Protocol Architecture for Application-Specific Networking**. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, pages 55–64, 1996. https://www.usenix.org/legacy/publications/library/proceedings/sd96/mef.html.

[40] Edoardo Biagioni, Robert Harper, and Peter Lee. **A Network Protocol Stack in Standard ML**. *Higher Order Symbol. Comput.*, 14(4):309–356, December 2001. ISSN 1388-3690. doi: 10.1023/A:1014403914699. https://doi.org/10.1023/A:1014403914699.

[41] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. **Signatures for a Network Protocol Stack: A Systems Application of Standard ML**. *SIGPLAN Lisp Pointers*, VII(3):55–64, July 1994. ISSN 1045-3563. doi: 10.1145/182590.182431. http://doi.acm.org/10.1145/182590.182431.

[42] Edoardo Biagioni. **A Structured TCP in Standard ML**. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 36–45, New York, NY, USA, 1994. ACM. ISBN 0-89791-682-4. doi: 10.1145/190314.190318. http://doi.acm.org/10.1145/190314.190318.

[43] Guangrui Fu. **Design and implementation of an operating system in Standard ML**. Master's thesis, University of Hawaii at Manoa, 1999. http://www2.hawaii.edu/~esb/prof/proj/hello/.

[44] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. **The Flux OSKit: A Substrate for Kernel and Language Research**. *SIGOPS Oper. Syst. Rev.*, 31 (5):38–51, October 1997. ISSN 0163-5980. doi: 10.1145/269005.266642. http://doi.acm.org/10.1145/269005.266642.

[45] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. **A Principled Approach to Operating System Construction in Haskell**. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086380. http://doi.acm.org/10.1145/1086365.1086380.

[46] Galen C. Hunt and James R. Larus. **Singularity: Rethinking the Software Stack**. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243424. http://doi.acm.org/10.1145/1243418.1243424.

[47] Jean Yang and Chris Hawblitzel. **Safe to the Last Instruction: Automated Verification of a Type-safe Operating System**. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 99–110, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806610. http://doi.acm.org/10.1145/1806596.1806610.

[48] Anil Madhavapeddy. **Combining Static Model Checking with Dynamic Enforcement Using the Statecall Policy Language**. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 446–465, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10372-8. doi: 10.1007/978-3-642-10373-5_23. http://dx.doi.org/10.1007/978-3-642-10373-5_23.

[49] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. **Melange: Creating a "Functional"Internet**. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 101–114, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273009. http://doi.acm.org/10.1145/1272996.1273009.

[50] Javier Paris, Victor Gulias, and Alberto Valderruten. **A High Performance Erlang TCP/IP Stack**. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, pages 52–61, New York, NY, USA, 2005. ACM. ISBN 1-59593-066-3. doi: 10.1145/1088361.1088372. http://doi.acm.org/10.1145/1088361.1088372.

[51] Cloudozer LLP. **LING: Erlang on Xen**, 2013. http://erlangonxen.org/.

[52] Martin Schoeberl. **ejIP: A TCP/IP Stack for Embedded Java**. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 63–69, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093167. http://doi.acm.org/10.1145/2093157.2093167.

[53] Vincent St-Amour, Lysiane Bouchard, and Marc Feeley. **Small Scheme Stack: A Scheme TCP/IP Stack Targeting Small Embedded Applications**. *2008 Workshop on Scheme and Functional Programming*, 2008. http://www.iro.umontreal.ca/~feeley/papers/StAmourBouchardFeeleySW08.pdf.

[54] Harshal Sheth and Aashish Welling. **An Implementation and Analysis of a Kernel Network Stack in Go with the CSP Style**. *CoRR*, abs/1603.05636, 2016. http://arxiv.org/abs/1603.05636.

[55] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. **The benefits and costs of writing a POSIX kernel in a high-level language**. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 89–105, Carlsbad, CA, 2018. USENIX As-

sociation. ISBN 978-1-931971-47-8. https://www.usenix.org/conference/osdi18/presentation/cutler.

[56] Google, Inc. **netstack: IPv4 and IPv6 userland network stack**, 2016. https://github.com/google/netstack.

[57] Google, Inc. **gVisor Container Runtime Sandbox**, 2018. https://github.com/google/gvisor.

[58] The Go Authors. **The Go Programming Language Documentation: Data Race Detector**, 2013. https://golang.org/doc/articles/race_detector.html.

[59] M. Paolino, N. Nikolaev, J. Fanguede, and D. Raho. **SnabbSwitch user space virtual switch benchmark and performance optimization for NFV**. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 86–92, Nov 2015. doi: 10.1109/NFV-SDN.2015.7387411. https://ieeexplore.ieee.org/document/7387411/.

[60] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. **RustBelt: Securing the Foundations of the Rust Programming Language**. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017. ISSN 2475-1421. doi: 10.1145/3158154. http://doi.acm.org/10.1145/3158154.

[61] Isaac Gouy. **The Computer Language Benchmarks Game**, 2018. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[62] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. **System Programming in Rust: Beyond Safety**. *SIGOPS Oper. Syst. Rev.*, 51(1):94–99, September 2017. ISSN 0163-5980. doi: 10.1145/3139645.3139660. http://doi.acm.org/10.1145/3139645.3139660.

[63] M-Labs. **smoltcp: smol TCP/IP stack in Rust**, 2016. https://github.com/m-labs/smoltcp.

[64] Redox Developers. **The Redox Operating System**. 2015. https://www.redox-os.org/.

[65] Galois, Inc. **rustwall: Rust firewall for seL4**, 2018. https://github.com/GaloisInc/rustwall.

[66] John Ericson. **QuiltOS: QuiltNet network stack in Rust**. 2016. https://github.com/QuiltOS/QuiltNet.

[67] Andrei Maximov. **usrnet: user space TCP/IP stack written in Rust**, 2018. https://github.com/andreimaximov/usrnet.

[68] The Fuchsia Authors. **Fuchisa recovery netstack**, 2018. `https://fuchsia.googlesource.com/garnet/+/master/bin/recovery_netstack/core/src/`.

[69] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. **The Case for Writing a Kernel in Rust**. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 1:1–1:7, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5197-3. doi: 10.1145/3124680.3124717. `http://doi.acm.org/10.1145/3124680.3124717`.

[70] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. **Multiprogramming a 64kB Computer Safely and Efficiently**. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132786. `http://doi.acm.org/10.1145/3132747.3132786`.

[71] Fredrik Nilsson and Niklas Adolfsson. **A Rust-based Runtime for the Internet of Things**. Master's thesis, University of Gothenburg and Chalmers University of Technology, 2017. `http://publications.lib.chalmers.se/records/fulltext/250074/250074.pdf`.

[72] Robert Clipsham. **Safe, Correct, and Fast Low-Level Networking**. Master's thesis, University of Glasgow, 2015. `https://csperkins.org/research/thesis-msci-clipsham.pdf`.

[73] Luigi Rizzo and Matteo Landi. **Netmap: Memory Mapped Access to Network Devices**. *SIGCOMM Comput. Commun. Rev.*, 41(4):422–423, August 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018500. `http://doi.acm.org/10.1145/2043164.2018500`.

[74] Robert Clipsham. **libpnet: Cross-platform, low level networking using Rust**, 2014. `https://github.com/libpnet/libpnet`.

[75] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. **NetBricks: Taking the V out of NFV**. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 203–216, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. `http://dl.acm.org/citation.cfm?id=3026877.3026894`.

[76] Data Plane Development Kit Project. **DPDK**, 2013. `http://www.dpdk.org/`.

[77] K. Lee, S. Woo, S. Seo, J. Park, S. Ryu, and S. Moon. **Toward building memory-safe network functions with modest performance overhead**. In *The Third Workshop on Networking and Programming Languages*, NetPL 2017. ACM SIGCOMM, 2017.

https://conferences.sigcomm.org/sigcomm/2017/files/program-netpl/sigcomm17netpl-paper5.pdf.

[78] Simon Ellmann. **Writing Network Drivers in Rust**. Bachelor's thesis, Technical University of Munich, 2018. https://www.net.in.tum.de/fileadmin/bibtex/publications/theses/2018-ixy-rust.pdf.

[79] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. **Implementing Network Protocols at User Level**. *IEEE/ACM Trans. Netw.*, 1(5):554–565, October 1993. ISSN 1063-6692. doi: 10.1109/90.251914. http://dx.doi.org/10.1109/90.251914.

[80] Chris Maeda and Brian N. Bershad. **Protocol Service Decomposition for High-performance Networking**. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 244–255, New York, NY, USA, 1993. ACM. ISBN 0-89791-632-8. doi: 10.1145/168619.168639. http://doi.acm.org/10.1145/168619.168639.

[81] Aled Edwards and Steve Muir. **Experiences Implementing a High Performance TCP in User-space**. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pages 196–205, New York, NY, USA, 1995. ACM. ISBN 0-89791-711-1. doi: 10.1145/217382.318122. http://doi.acm.org/10.1145/217382.318122.

[82] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and C. Dalton. **User-space Protocols Deliver High Performance to Applications on a Low-cost Gb/s LAN**. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 14–23, New York, NY, USA, 1994. ACM. ISBN 0-89791-682-4. doi: 10.1145/190314.190316. http://doi.acm.org/10.1145/190314.190316.

[83] T. von Eicken, A. Basu, V. Buch, and W. Vogels. **U-Net: A User-level Network Interface for Parallel and Distributed Computing**. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224061. http://doi.acm.org/10.1145/224056.224061.

[84] Matt Welsh, Anindya Basu, and Thorsten von Eicken. **ATM and Fast Ethernet Network Interfaces for User-Level Communication**. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97), San Antonio, Texas, USA, February 1-5, 1997*, pages 332–342, 1997. doi: 10.1109/HPCA.1997.569697. https://doi.org/10.1109/HPCA.1997.569697.

[85] Matt Welsh, Anindya Basu, and Thorsten von Eicken. **Low-latency communication over Fast Ethernet**. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors,

Euro-Par'96 Parallel Processing, pages 185–194, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70633-5. https://link.springer.com/chapter/10.1007/3-540-61626-8_24.

[86] Matt Welsh, David Oppenheimer, and David Culler. **U-Net/SLE: A Java-based user-customizable virtual network interface**. *Scientific Programming*, 7(2):147–156, 1999. https://people.eecs.berkeley.edu/~culler/papers/unetsle.pdf.

[87] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. **ASHs: Application-specific Handlers for High-performance Messaging**. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 40–52, New York, NY, USA, 1996. ACM. ISBN 0-89791-790-1. doi: 10.1145/248156.248161. http://doi.acm.org/10.1145/248156.248161.

[88] Marc E. Fiuczynski, Richard P. Martin, Tsutomu Owa, and Brian N. Bershad. **SPINE: A Safe Programmable and Integrated Network Environment**. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, EW 8, pages 7–12, New York, NY, USA, 1998. ACM. doi: 10.1145/319195.319197. http://doi.acm.org/10.1145/319195.319197.

[89] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. **Virtual-Memory-Mapped Network Interfaces**. *IEEE Micro*, 15(1):21–28, 1995. doi: 10.1109/40.342014. https://doi.org/10.1109/40.342014.

[90] Compaq/Intel/Microsoft. **Virtual Interface Architecture Specification, Version 1.0**. 1997. http://www.cs.uml.edu/~bill/cs560/VI_spec.pdf.

[91] NERSC, PC Cluster Project. **M-VIA: A High Performance Modular VIA for Linux**, 2001. https://linas.org/mirrors/www.nersc.gov/2001.02.13/research/FTG/via/.

[92] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai Li. **Experiences with VI Communication for Database Storage**. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 257–268, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1605-X. http://dl.acm.org/citation.cfm?id=545215.545244.

[93] Norman C. Hutchinson and Larry L. Peterson. **The *x*-Kernel: An Architecture for Implementing Network Protocols**. *IEEE Trans. Software Eng.*, 17(1):64–76, 1991. doi: 10.1109/32.67579. https://doi.org/10.1109/32.67579.

[94] Torsten Braun, Christophe Diot, Anna Hoglander, and Vincent Roca. **An Experimental User Level Implementation of TCP**. Technical Report RR-2650, INRIA, September 1995. https://hal.inria.fr/inria-00074040.

[95] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. **Exokernel: An Operating System Architecture for Application-level Resource Management**. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224076. http://doi.acm.org/10.1145/224056.224076.

[96] Dawson R. Engler and M. Frans Kaashoek. **DPF: Fast, Flexible Message Demultiplexing Using Dynamic Code Generation**. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 53–59, New York, NY, USA, 1996. ACM. ISBN 0-89791-790-1. doi: 10.1145/248156.248162. http://doi.acm.org/10.1145/248156.248162.

[97] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. **Fast and Flexible Application-level Networking on Exokernel Systems**. *ACM Trans. Comput. Syst.*, 20(1):49–83, February 2002. ISSN 0734-2071. doi: 10.1145/505452.505455. http://doi.acm.org/10.1145/505452.505455.

[98] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. **The design and implementation of an operating system to support distributed multimedia applications**. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sep 1996. ISSN 0733-8716. doi: 10.1109/49.536480. https://ieeexplore.ieee.org/document/536480.

[99] Richard Black, Paul T. Barham, Austin Donnelly, and Neil Stratford. **Protocol Implementation in a Vertically Structured Operating System**. In *Proceedings of the 22Nd Annual IEEE Conference on Local Computer Networks*, LCN '97, pages 179–188, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8141-1. http://dl.acm.org/citation.cfm?id=648046.745222.

[100] Adam Dunkels. **Full TCP/IP for 8-bit Architectures**. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 85–98, New York, NY, USA, 2003. ACM. doi: 10.1145/1066116.1066118. http://doi.acm.org/10.1145/1066116.1066118.

[101] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. **The Multikernel: A New OS Architecture for Scalable Multicore Systems**. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44,

New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629579. http://doi.acm.org/10.1145/1629575.1629579.

[102] Joan Lledó. **GNU HURD: LwIP translator**, 2017. http://lists.gnu.org/archive/html/bug-hurd/2017-08/msg00035.html.

[103] Free Software Foundation, Inc. **GNU HURD: subhurd**, 2017. http://www.gnu.org/software/hurd/hurd/subhurd.html.

[104] Genode Labs GmbH. **Release notes for the Genode OS Framework 18.08: New VFS plugin for using LwIP as TCP/IP stack**, 2018. https://genode.org/documentation/release-notes/18.08#New_VFS_plugin_for_using_LwIP_as_TCP_IP_stack.

[105] J. Mogul, R. Rashid, and M. Accetta. **The Packer Filter: An Efficient Mechanism for User-level Network Code**. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 39–51, New York, NY, USA, 1987. ACM. ISBN 0-89791-242-X. doi: 10.1145/41457.37505. http://doi.acm.org/10.1145/41457.37505.

[106] Steven McCanne and Van Jacobson. **The BSD Packet Filter: A New Architecture for User-level Packet Capture**. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association. http://dl.acm.org/citation.cfm?id=1267303.1267305.

[107] Wen Xu and Anees Shaikh. **Daytona: A User-Level TCP Stack**. 2002. http://nms.lcs.mit.edu/~kandula/data/daytona.pdf.

[108] David Ely, Stefan Savage, and David Wetherall. **Alpine: A User-level Infrastructure for Network Protocol Development**. In *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 3*, USITS'01, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association. http://dl.acm.org/citation.cfm?id=1251440.1251455.

[109] Jamal Hadi Salim. **TC Classifier Action Subsystem Architecture**, 2015. https://web.archive.org/web/20160330005952/netdev01.org/sessions/21.

[110] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. **The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel**. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6080-7. doi: 10.1145/3281411.3281443. https://doi.org/10.1145/3281411.3281443.

[111] Björn Töpel and Magnus Karlsson. **Linux NetDev RFC: Introducing AF_XDP support**, 2018. https://patchwork.ozlabs.org/cover/867937/.

[112] Björn Töpel and Magnus Karlsson. **FOSDEM 2018 Talk: Fast packet processing in Linux with AF_XDP**, 2018. https://fosdem.org/2018/schedule/event/af_xdp/.

[113] Cilium Authors. **Cilium software**, 2018. https://github.com/cilium/cilium.

[114] William Tu. **AF_XDP support for OVS**, 2018. https://mail.openvswitch.org/pipermail/ovs-dev/2018-August/351295.html.

[115] ntop. **PF_RING**, 2004. https://www.ntop.org/products/packet-capture/pf_ring/.

[116] N. Bonelli, S. Giordano, and G. Procissi. **Network Traffic Processing With PFQ**. *IEEE Journal on Selected Areas in Communications*, 34(6):1819–1833, June 2016. ISSN 0733-8716. doi: 10.1109/JSAC.2016.2558998. https://ieeexplore.ieee.org/document/7460204.

[117] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni. **A purely functional approach to packet processing**. In *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 219–230, Oct 2014.

[118] Linux Foundation. **Open vSwitch**, 2009. http://www.openvswitch.org/.

[119] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. **SoftNIC: A Software NIC to Augment Hardware**. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[120] Julian Stecklina. **Shrinking the Hypervisor One Subsystem at a Time: A Userspace Packet Switch for Virtual Machines**. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 189–200, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2764-0. doi: 10.1145/2576195.2576202. http://doi.acm.org/10.1145/2576195.2576202.

[121] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. **NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms**. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, 2014. USENIX Association. ISBN 978-1-931971-09-6. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang.

[122] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. **mSwitch: A Highly-scalable, Modular Software Switch**. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 1:1–1:13, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3451-8. doi: 10.1145/2774993.2775065. http://doi.acm.org/10.1145/2774993.2775065.

[123] Yutaro Hayakawa. **VALE-bpf: VALE eBPF extension module**, 2017. https://github.com/YutaroHayakawa/vale-bpf.

[124] Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and Michio Honda. **HyperNF: Building a High Performance, High Utilization and Fair NFV Platform**. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 157–169, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5028-0. doi: 10.1145/3127479.3127489. http://doi.acm.org/10.1145/3127479.3127489.

[125] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. **ClickOS and the Art of Network Function Virtualization**. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, 2014. USENIX Association. ISBN 978-1-931971-09-6. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins.

[126] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. **Rekindling Network Protocol Innovation with User-level Stacks**. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, April 2014. ISSN 0146-4833. doi: 10.1145/2602204.2602212. http://doi.acm.org/10.1145/2602204.2602212.

[127] Martin Unzner. **A Split TCP/IP Stack Implementation for GNU/Linux**. Master's thesis, TU Dresden, 2014. http://os.inf.tu-dresden.de/papers_ps/unzner-diplom.pdf.

[128] NetBSD Kernel Interfaces Manual. **SHMIF(4): shmif – rump shared memory network interface**, 2010. http://netbsd.gw.com/cgi-bin/man-cgi?shmif+4+NetBSD-7.0.

[129] Ian A. Pratt and Keir Fraser. **Arsenic: A User-Accessible Gigabit Ethernet Interface**. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Twenty years into the communications odyssey, Anchorage, Alaska, USA, April 22-26, 2001*, pages 67–76, 2001. doi: 10.1109/INFCOM.2001.916688. https://doi.org/10.1109/INFCOM.2001.916688.

[130] Solarflare. **OpenOnload**, 2002. http://www.openonload.org.

[131] Mao Miao, Xiaohui Luo, Fengyuan Ren, Wenxue Cheng, Jing Xie, Wenzhuo Li, and Xiaolan Liu. **Renovate high performance user-level stacks' innovation utilizing commodity network adaptors**. In *2017 IEEE Symposium on Computers and Communications (ISCC)*, pages 906–911, July 2017. doi: 10.1109/ISCC.2017.8024641. https://ieeexplore.ieee.org/document/8024641.

[132] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. **Arrakis: The Operating System is the Control Plane**. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. http://dl.acm.org/citation.cfm?id=2685048.2685050.

[133] Yukai Huang, Jinkun Geng, Du Lin, Bin Wang, Junfeng Li, Ruilin Ling, and Dan Li. **LOS: A High Performance and Compatible User-level Network Operating System**. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, pages 50–56, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5244-4. doi: 10.1145/3106989.3106997. http://doi.acm.org/10.1145/3106989.3106997.

[134] Hajime Tazaki, Ryo Nakamura, and Yuji Sekiya. **Library Operating System with Mainline Linux Network Stack**. *netdev 2015*, 2015. https://people.netfilter.org/pablo/netdev0.1/papers/Library-Operating-System-with-Mainline-Linux-Network-Stack.pdf.

[135] Patrick Kelsey. **libuinet: A library version of FreeBSD's TCP/IP stack plus extras**, 2014. https://github.com/pkelsey/libuinet.

[136] Tencent. **F-Stack (DPDK and FreeBSD TCP/IP)**, 2017. https://github.com/F-Stack/f-stack.

[137] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. **mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems**. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. http://dl.acm.org/citation.cfm?id=2616448.2616493.

[138] Ilias Marinos, Robert N.M. Watson, and Mark Handley. **Network Stack Specialization for Performance**. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626311. http://doi.acm.org/10.1145/2619239.2626311.

[139] ScyllaDB. **SeaStar**, 2014. http://www.seastar-project.org/.

[140] M. S. Thompson, A. S. Abdallah, J. M. Reed, A. B. Mackenzie, and L. A. Dasilva. **The FINS framework: an open source userspace networking subsystem for Linux**. *IEEE Network*, 28(5):32–37, September 2014. ISSN 0890-8044. doi: 10.1109/MNET.2014.6915437. https://ieeexplore.ieee.org/abstract/document/6915437.

[141] Jon Howell, Bryan Parno, and John R. Douceur. **How to Run POSIX Apps in a Minimal Picoprocess**. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 321–332, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-01-0. https://www.usenix.org/conference/atc13/technical-sessions/presentation/howell.

[142] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. **IX: A Protected Dataplane Operating System for High Throughput and Low Latency**. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. http://dl.acm.org/citation.cfm?id=2685048.2685053.

[143] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. **The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane**. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, December 2016. ISSN 0734-2071. doi: 10.1145/2997641. http://doi.acm.org/10.1145/2997641.

[144] George Prekas, Marios Kogias, and Edouard Bugnion. **ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks**. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132780. http://doi.acm.org/10.1145/3132747.3132780.

[145] Ali Raza. **UKL: A Unikernel Based on Linux**, 2018. https://next.redhat.com/2018/11/14/ukl-a-unikernel-based-on-linux/.

[146] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. **Unikernels As Processes**. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 199–211, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267845. http://doi.acm.org/10.1145/3267809.3267845.

[147] Takayuki Imada. **MirageOS Unikernel with Network Acceleration for IoT Cloud Environments**. In *Proceedings of the 2018 2nd International Conference on Cloud and Big Data Computing*, ICCBDC'18, pages 1–5, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6474-4. doi: 10.1145/3264560.3264561. http://doi.acm.org/10.1145/3264560.3264561.

[148] Luigi Rizzo and Giuseppe Lettieri. **VALE, a Switched Ethernet for Virtual Machines**. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1775-7. doi: 10.1145/2413176.2413185. http://doi.acm.org/10.1145/2413176.2413185.

[149] Pierre Krieger. **tiny-http: Low level HTTP server library in Rust**, 2014. https://github.com/tiny-http/tiny-http.

[150] The Rouille Developers. **rouille: Web framework in Rust**, 2016. https://github.com/tomaka/rouille.

[151] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. **Nethammer: Inducing Rowhammer Faults through Network Requests**. *CoRR*, abs/1805.04956, 2018. http://arxiv.org/abs/1805.04956.

[152] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. **Throwhammer: Rowhammer Attacks over the Network and Defenses**. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. https://www.usenix.org/conference/atc18/presentation/tatar.

Sources were last accessed on December 4, 2018. The year refers to the publication. Captures of sources without DOI, ACM, USENIX, IEEE, Springer, arXiv.org, or GitHub URLs are available at the Internet Archive Wayback Machine via the prefix http://web.archive.org/web/2018120/ *followed directly by the full URL.*