Byways: High-Performance, Isolated Network Functions for Multi-Tenant Cloud Servers

Xinyu Han
The George Washington University
Washington, D.C., USA
kevin_han@gwu.edu

Yuan Gao
The George Washington University
Washington, D.C., USA
ygao1@gwu.edu

Gabriel Parmer
The George Washington University
Washington, D.C., USA
gparmer@gwu.edu

Timothy Wood
The George Washington University
Washington, D.C., USA
timwood@gwu.edu

Abstract

Network functions (NFs) have become pervasive in data centers as a means to monitor and transform traffic as it flows between services. Softwarization of the network has further added to the diversity of functions that can be deployed, yet managing the performance, efficiency, tenant-customizability, and security of these functions remains a major challenge. We present Byways, an abstraction that provides facilitates to safely deploy NFs alongside end-host VMs in a multi-tenant cloud environment. Byways guarantee strict isolation between the host system, the network functions, and VM-based cloud applications, while still maintaining high performance. A Byway manages a specific set of services, and an associated NF only processes flows associated with those services, using per-byway resources (*e.g.,* processing time). This separation of end-host traffic across Byways provides strong fault isolation – a failing NF does not impact other services. Byways augment this isolation with per-Byway access rights that restrict a NFs access (*e.g.,* read, write, drop) to the flow, limiting the impact of a faulty NF on even its own services.

We have implemented BywayOS, a μ-kernel instantiation of Byways, and evaluated its performance, efficiency, and isolation properties compared to state of the art virtual machine networking technologies. A Byway processing memcached traffic through an isolated NF demonstrates throughput and latency competitive with, and often better than, Linux host performance (*i.e.,* without a NF nor a VM), and throughput 1.25x-6.43x higher than other host NF+VM technologies, while offering stronger isolation and a trusted computing base (TCB) more than 20x smaller.

CCS Concepts

• **Networks** → **Cloud computing**.

Keywords

Cloud computing, Network function, Multi-tenant, End host, Resource isolation

ACM Reference Format:
Xinyu Han, Yuan Gao, Gabriel Parmer, and Timothy Wood. 2024. Byways: High-Performance, Isolated Network Functions for Multi-Tenant Cloud Servers. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3698038.3698547

1 Introduction

Network Functions (NFs) are essential in the modern cloud infrastructure. While these functions originated as a means for network operators to provide services transparently to clients and servers, there is a growing desire from both infrastructure providers and application owners to deploy their own NFs [27, 63]. New deployment models like microservices and serverless computing require a variety of network functionality including proxies, network monitoring, and load balancers. As a result, NFs are becoming more application specific, and tenants within a cloud infrastructure desire ways to dynamically manage the mapping of flows to their own collection of network functions and end-host applications.

Middlebox NF processing (*i.e.,* running on specialized servers acting as a "bump in the wire") has received a lot of attention [42, 48, 49, 82], but these approaches can see significant performance overheads if deployed on top of existing virtualization layers or in cloud environments where tenants lack control to determine where functions are deployed. On

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698547

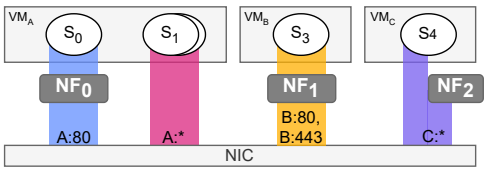


Figure 1: Byways **connect services hosted inside VMs to the network. A** Byway **can be configured to direct traffic for particular IP:port(s) pairs to a NF (*e.g.,* $A{:}80$ goes to NF_0 before going to service S_0). Unmanaged traffic (*e.g.,* $A:*$ for $s_1 - s_2$) goes directly to the VM. NF access rights can be limited, *e.g.,* NF_2 can observe traffic to S_4, but cannot disrupt it.**

the other hand, end-host NFs are pervasive: from firewalls, programmable logic via eBPF [16], user-level processing [47], and service and network virtualization [15, 68], to intrusion detection and prevention [64]. End-host based NFs offer application specific services [3, 24], and since they execute on the host resources, accounting and billing can be more easily charged to the tenants. Application-specific, end-host NF processing can improve security [7], improves resource utilization and decreases costs [26, 83], facilitates scalable execution [26, 84], manages latency and application QoS properties [3, 7, 83], and can aid in managing limited edge resources [59] with low latency [61].

End-host NFs require system support for execution. This involves either the NF's execution within a tenant's VM (*e.g.,* with Linux `iptables` or NFQUEUE [47]), or in an environment tightly controlled by the cloud provider (*e.g.,* with OpenVSwitch [54] or VPP [4]). The existing solutions for both of these options are problematic. When deployed in a tenant VM, the NF and host OS must trust each other. When deployed independent of a tenant VM, it is more challenging to specialize the NF, and the tenant must trust the NF to properly process all its traffic. Both options are ill-suited for the fact that *NFs, like other trusted infrastructure, can fail or be compromised* [28, 72, 73, 83] (in §2.1, see a small subset of NF CVEs).

End-host NF infrastructure is in a difficult design space. It must provide NF execution that is (1) high-performance, (2) strongly isolated from tenant failures, and (3) customizable for individual tenants, applications, and services. Additionally, tenants should suffer graceful degradation when NFs fail. While a failure in a NF will impact the services whose flows it is processing, it should *not* impact other services nor tenants. In short, what is necessary is a strong *multi-tenant infrastructure* for NF execution [3].

To address this challenge, this paper introduces the Byway abstraction for multi-tenant NF execution. Byways enable the strong isolation of NFs from tenant VMs, and vice-versa. The Byway abstraction provides a devoted path for a VM's service's network flows that are processed by a specific NF, as illustrated in Figure 1. Byways are designed to minimize the potential negative impact of a NF on its associated service, and to tightly limit the impact of NF faults to only that service. As such, Byways are *service-specific* and only process their associated service's flows, and enable *restricted access*, limiting a NF's ability to impact a service's flows by controlling NF's access rights to the flows.

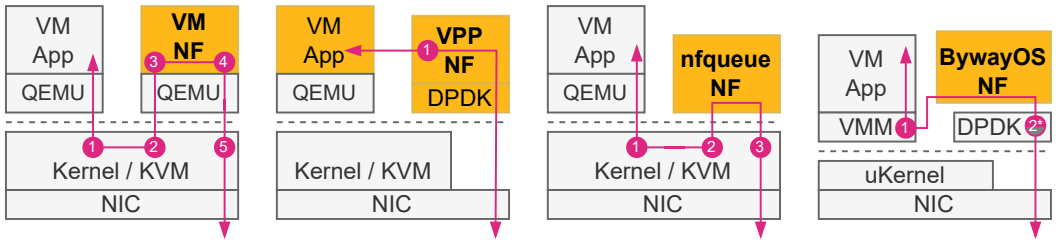
Type	NF Software	CVE-*
DoS	Snort, Envoy, Cilium, OVS, StrongSwan, OpenSSL, HAProxy	2023-{20270, 20083, 35945, 27496, 41333, 6129, 5678, 25950, 0056}, 2022-{31045, 29178, 40617, 20751}, 2021-{3905, 40114}, 2019-25076, 2018-{17204, 5388}, 2015-3991
Data leak	Snort, Suricata, Cilium, nDPI, HAProxy	2023-{34242, 45539, 0836, 25950}, 2021-34749, 2020-{19678, 11939}
CFI viol.	Snort, Suricata, OVS, nDPI, Strongswan	2023-{35853, 41913, 26463}, 2021-34749, 2020-11939, 2016-2074
Priv. Escalation	Envoy, eBPF, OpenSSL, HAProxy	2023-{27488, 39191}, 2022-25218, 2020-35195

Table 1: Network function security vulnerabilities.

Byways contributions include enhancing NF isolation across three dimensions, without sacrificing performance nor legacy support:

- System Isolation: Byways enable a multi-tenant NF execution model that insulates NFs, VMs, and the host from each other through a per-tenant, microkernel based virtual machine monitor and efficient communication channels.
- Traffic Isolation: the Byways abstraction restricts a NF to process only a subset of a VM's network flows with a configurable set of capabilities following the principle of least privilege (PoLP).
- Temporal Isolation: The Byways scheduler separately, and preemptively allocates time to network functions to prevent timing attacks and denial of service attacks, while accounting execution costs to the appropriate tenant.

Lastly, we detail BywayOS, a μ-kernel based OS that enables tenant VM execution, and implements the Byway abstraction for NFs. While Byways could potentially be implemented in other infrastructures, BywayOS demonstrates that Byways can provide strong performance, while enjoying a high-confidence implementation due to a small TCB. Our evaluation demonstrates that BywayOS prevents a malicious or errant NF from negatively impacting VM service outside of those flows it is given access to. Our Byway communication abstraction is highly optimized, and provides performance better than solutions that offer no isolation – 25% higher memcached throughput and 68% lower tail latency versus VPP [4]. Compared to running network functions in NFQUEUE [47] with similar isolation properties, Byways achieve performance 2.9x / 6.4x higher on nginx and memcached.



(a) VM-based NF (b) VPP (c) NFQUEUE (d) BywayOS

Figure 2: Communication flow and packet copies (circles) for different approaches to host-based NF deployments. Shaded areas represent the components which can become compromised due to a NF failure or exploit. BywayOS **incurs two packet copies on the receive path, but can avoid the DPDK copy on transmission, giving it similar overhead to VPP, but with stronger isolation and a smaller TCB.**

System	NFs isolated from service VM	Per-VM NFs	Service-specific NFs	Service availability	Service confidentiality	Legacy NFs	High-perf
NFs-in-VM	●	●	○	○	○	●	○
VPP [4]	●	◐	○	○	○	◐	●
MTS [72]	●	●	○	○	○	●	◐
NFQUEUE [47]	●	●	●	●	●	◐	○
Netbricks [49]	●	●	○	○	●	○	●
Host eBPF [16]	●	●	○	○	○	○	●
BywayOS	●	●	●	●	●	◐	●

Table 2: Analysis of the isolation properties of NF systems. ●: supports, ○: does not support, and ◐: partial support.

2 Host NF Background & Related Work

NFs are a core part of data-center infrastructure and are often optimized to maintain line-rate performance. This performance optimization often leaves isolation, reliability, and security facilities lacking. For example: to enable zero-copy processing, packets are often placed in shared memory, which can lead to illicit access across domains [48, 82]; NFs might be directly activated via function calls upon packet reception, which can grant them direct access to packet memory even when not necessary [4]; and even NFs executed using safe languages [49] synchronously execute, which can delay processing in other NFs due to bugs or malicious attacks.

2.1 CVE analysis

NFs and NF frameworks are complex and network-facing, thus are susceptible to compromise. Table 1 presents a set of NF and NF framework compromises. Illustrative examples follow.

- *DoS.* CVE-2023-20083, CVE-2018-0230, and CVE-2006-6931 lead to exhaustion of CPU, preventing system response. Similarly, CVE-2022-20751, CVE-2021-40114, CVE-2023-35945, CVE-2021-4204, and CVE-2021-3905 lead to system memory exhaustion.
- *Privilege escalation.* CVE-2020-11939 and CVE-2020-11939 show examples of arbitrary code execution through CFI vulnerabilities. CVE-2023-33869, CVE-2023-39191, and CVE-2022-0500 demonstrate that such access can be leveraged to get root privilege. Potentially all of the CFI violations in Table 1 could similarly be extended to achieve privilege escalation.
- *Data leakage.* CVE-2023-25950, CVE-2021-34749, and CVE-2023-34242 feature NFs leaking sensitive information to the outside world as NFs are allowed to transmit-by-default, and to transmit to any addresses.

2.2 Host NF Solution Comparison

Here we describe current approaches for running network functions, with a focus on deployments that allow NFs to run safely on the same host as the applications sending or receiving the traffic. Table 2 depicts various systems for host NF execution discussed in further detail below. We qualitatively assess: (1) if the NFs are isolated from potentially disruptive actions by their service VMs, (2) if the system supports unique NFs per-VM, (3) if NFs can be constrained to process only traffic for specific services (*e.g.*, based on port number), (4) if VMs have guaranteed availability of packets regardless of NF behavior (*e.g.*, a crash), (5) if the platform can provide confidentiality such that NFs are unable to exfiltrate information that has been observed in a service's traffic, (6) if legacy NFs can be supported, and (7) if the NF infrastructure ensures high performance networking.

VM-based NFs. The simplest deployment of network functions is in virtual machines running alongside the

tenant's own application VMs. While this keeps the NF strongly isolated in its own virtual machine, it can incur a high performance cost due to the large number of packet copies incurred when traversing system boundaries. As shown in Figure 2(a), a simple VM-based function that uses the kernel networking stack will incur five copies for each packet sent or received from the application VM (three within the virtual machine monitor and two in and out of userspace for the NF VM). This cost becomes untenable when running low latency services, and there is no easy way to differentiate traffic for different services or ensure the confidentiality or availability of their traffic.

Software Switch-based NFs. High performance software switches such as OpenVSwitch (OVS) [54], Vector Packet Processing (VPP) [4], and Click [35, 43] provide efficient ways to route packets to different virtual machines. They provide core plugins that form a high-performance foundation for the Container Network Interface (CNI) [9] used in container orchestration environments such as Kubernetes [36]. We focus on VPP as a representative technology that provides an execution environment for NFs that efficiently operate on batches of packets before they are delivered to VMs. VPP executes NFs in user space with direct access to the NIC using DPDK [32]. Those NFs can act as an intermediary for network processing to a VM using `vhost_user`. As such, VPP has direct access to the VM's memory, thus can transfer packets to and from the VM quickly, albeit with no fault isolation. VPP's structure is shown in Figure 2(b).

VPP's direct NF access to both the NIC driver and to the VM's memory limits the reliability of NF execution: (1) the lack of memory isolation makes NF-runtime, or VM corruption trivial; (2) the execution of NFs synchronously (via function call) means that a NF that uses an undue amount of CPU time can break availability; and (3) because all network traffic goes through the VPP runtime, VMs cannot enforce a distinction between traffic which should be processed by the NF and that which should be isolated from it.

The standard deployment of VPP means that there is isolation between neither the NF and the VM, nor between NFs run by different tenants since all are within a single runtime with shared memory access across VMs. In order to provide stronger isolation, the NIC hardware can present multiple virtual networking devices using SR-IOV [65]. In our evaluation we use this with VPP to enable a separate VPP instance to run per-VM NFs specific to each tenant. With this setup, each VM must trust their NFs in the hope that there will be no bugs or compromises, but at least there is spatial separation between the NFs for each VM. Other NFV research platforms such as E2, ClickOS, and OpenNetVM have comparable designs to VPP, leading to similar performance and isolation properties [48, 82].

MTS [72] seeks to improve isolation by deploying a copy of OVS for each tenant. With this design, the routing infrastructure for each tenant is isolated, but MTS does not directly provide support for running network functions. Network functions could be run within separate VMs and managed by the tenant-specific OVS, to provide high customization of per-VM, service specific NFs. However, NFs can still disrupt the data flow to application VMs, and the system does not ensure high performance of NFs, only of the OVS infrastructure. Additionally, NF/VM coordination uses SR-IOV, thus the NF can trivally be bypassed by not addressing its endpoint. Thus, we use VPP as our key comparison for this class of NF infrastructures.

Netfilter Queue (NFQUEUE). An examplar of approaches that aim to provide stronger isolation properties to VMs by tightening the capabilities of the NF runtime is Linux's NFQUEUE, depicted in Figure 2(c). The kernel's device drivers are used to process network packets, and *iptables* is used to flexibly route them either to a NFQUEUE, or to the VM. Packets forwarded to NFQUEUE are enqueued to a user-level process containing the NF logic. That NF can operate on the packet, and forward the output to the VM. Comparably, when the VM transmits a packet, the NFQUEUE NF process interposes on the network transmission.

NFQUEUE performance is limited by the processing and communication pathways from the host network stack to the NF, and the NF to the VM. Context switching, mode transitions, and IPC logic limit the overall throughput of NF processing. While Linux IPC is often sufficient for many tasks, per-packet IPC can be a bottleneck (at more than 2μsec an IPC [22]). Similar to VPP, each tenant VM's traffic can be processed by separate NFs, but unlike VPP, specific flows can be directed to different NFs or bypass them entirely. Thus, a partially trusted NF (because it is potentially faulty or susceptible to compromise) can be limited to only process on and impact a subset of a VM's networking traffic. As such, NFQUEUE offers capabilities for heightened isolation of the VM from the NF, at the cost of performance.

memcached performance	NFQUEUE	VPP	BywayOS
Throughput (reqs/sec)	15.5K	79.9K	107.8K
P99 Latency (ms)	5.5	2.9	0.75

Table 3: memcached performance with NF technologies.

To understand the performance of NFQUEUE, VPP, and BywayOS, Table 3 displays peak memcached performance results (details in §5). NFQUEUE's heightened isolation comes at the cost of performance, while VPP achieves better performance at the cost of isolation. BywayOS's design enables both strong performance and strict isolation.

Language safety for NF execution. Middlebox infrastructures such as NetBricks [49] and SafeBricks [56] augment kernel-bypass I/O using DPDK with NFs written in safe languages. NFs are limited by the language to access only

packets and data-structures explicitly passed to the NF, preventing some types of attacks. However, this approach requires rewriting software in a safe language, which limits legacy support. The performance strengths of direct synchronous execution via function calls is also a potential liability as a NF that does not return promptly could delay servicing other packets either due to malicious attacks or bugs. We seek to provide a solution that offers an easier path to legacy support while also enabling the task preemption that is needed to prevent this type of timing attack.

Extended Berkeley Packet Filter (eBPF) NFs. eBPF [16] programs hook into Linux kernel execution paths, and are executed in a sandbox environment. Network functions can be built using XDP [30] hooks in the lower-levels of the Linux networking stack to trigger computation on packets. The eBPF sandbox in the kernel ensures type and memory safety of programs, but cannot support legacy NFs and has a more limited programming model than the languages typically used for NF development.

Access to eBPF is generally restricted to trusted parties. While the eBPF sandbox provides type and memory safety, it exposes a large number of kernel-internal APIs, and it is not available for unprivileged use such as by cloud tenants. Linux capabilities for `CAP_BPF` are necessary to insert eBPF programs, and XDP [30] programs require both `CAP_NET_ADMIN` and `CAP_SYS_ADMIN`. With these capabilities, a user can generally configure network settings including firewalls, and modify fundamental namespaces in networking and the filesystem, changing the hostname, update resource allocations (through `rlimit`), and many other functions often associated with `root` privileges. An ongoing concern is whether eBPF programs running in the same hardware memory protection as the rest of the kernel, can leverage speculative execution attacks such as Spectre [34] to leak kernel data. As such, eBPF does not currently provide a solution for running untrusted NFs. eBPF is often paired with a user-level solution such as VPP (as in Calico [6]) to run legacy NFs or code that otherwise isn't a fit for the BPF sandbox.

Hardware-based Functions. Several cloud providers have embraced deploying their network functions on specialized hardware such as programmable SmartNICs [2, 19]. This very strongly isolates network functions from the tenant VMs, but the NIC platform is not available for tenants to utilize for their own NFs. Recent work has sought ways to support multi-tenancy on SmartNICs, but they still cannot be used to deploy complex functions or ensure strong fault isolation. SuperNIC [38] allows multi-tenant functions on a NIC, but requires them to be deployed to FPGAs, vastly increasing the difficulty of deploying legacy functions. FairNIC [27] allows NFs to run on System-on-a-Chip based NICs, but its focus is on ensuring performance fairness, and cannot provide any form of security between functions. While S-NIC [83] offers both NF memory and performance isolation on SmartNICs, it neither considers fine-grained isolation properties, nor provides service level solutions for VM tenants.

3 Byways **Design**

Byways are designed for a complicated multi-tenancy model where different tenants execute (VMs and NFs) on the same shared hardware. This is a critical model for cloud environments where, for example, Amazon uses virtualization techniques such as Xen [11], KVM [39], or Firecracker [1, 2] to provide secure isolation between tenant applications. Byways extend this with a powerful NF execution model to provide the complex and necessary network services required in the modern cloud. Given that buggy or compromised NFs can easily threaten the proper execution of a system (§2), it is important to strongly constrain their access to service's networking flows. We aim to decouple the NF software from that of the cloud provider and the tenant, so that both can safely interact with only partially trusted NFs. This enables an app-store equivalent for NFs, rather than rely on centralized deployment of NFs in middleboxes run by cloud operators. The challenge with a multi-tenant NF infrastructure is that there are *explicit dependencies between tenant services and NFs that process their traffic.* As such, Byways aim to tightly control the access rights for NFs according to their goals, thus limiting the scope of impact of faulty, malicious, or compromised NFs on VMs.

The design of Byways focuses around key properties:

- *Per-service data channels (§3.1).* Network functions interpose on the traffic for only a restricted set of services within a VM, as part of a channel from NIC to service. All traffic flows destined for, or derived from, the service traverse the channel. As such, a faulty NF in a Byway for an HTTP server cannot impact traffic for `ssh`, even if in the same VM. These channels are designed to efficiently and securely shuttle packets between NIC driver, NF, VMM, and to the service in the VM by ensuring that the packets sent into the VM are correctly addressed to the service. Services that don't require NF processing are associated with an "unmanaged" Byway. *NFs only process and can impact data for their associated services.*
- *Restricted NF data channel access rights (§3.2).* Network functions are granted only limited access to the Byway's data channel. NF's ability to threaten the integrity, confidentiality, and availability of channel data is intentionally restricted. For example, a firewall might have permission to drop a service's incoming packets, but not to modify them nor access outgoing packets. *NFs can only impact service flows as much as is necessary for NF goals (PoLP).*

- *Safe NF execution (§3.3).* Each NF executes in a separate memory protection domain, with minimal access to external APIs, and with preemptively scheduled execution. As such, NFs run in a tightly constrained, "share nothing", execution environment augmented only with access to data channels. A NF that uses undue resources can be separately throttled. *NFs are tightly memory- and processing-restricted to limit their ability to interfere with other NFs and VMs beyond their data channels.*

Given these properties, Byways result in a *system structure* on the end-host that matches the structure of *network flows*. Further, our design expands on these properties to strengthen the security of the system. This includes two core requirements:

- *Service multi-tenancy through VMs (§3.4).* Byways aim to be implemented in a conventional, strong multi-tenant model for different services that underlies the current cloud. This typically requires VM-based isolation between tenants (*e.g.,* Amazon Functions backed by Firecracker VMs [1]) as the infrastructure must maintain isolation even when tenant's kernels are compromised. VMs are provided by minimal VMMs that understand which NFs communicate through which Byways, with which allowed ports. VMs provide strong service tenant isolation, and VMM integration into Byways is integral in maintaining the above properties.
- *Minimal shared system services and trusted computing base (TCB).* The underlying system that provides the above properties must do so in a trustworthy manner. As such, our design focuses on implementing Byways with a minimal trusted computing base, and avoiding sharing services between tenants. The former increases confidence that system compromises are avoided, and the latter mitigates the scope of impact of compromises that do happen.

3.1 Byway **Data Channels**

Byways define the path that packet flows take to reach a VM-based service. Figure 1 depicts Byways connecting various services to the network. It illustrates how Byways are restricted to flows associated with a set of ports, allowing packets properly associated with the allowed ports to reach the NIC and the VM. The Byway to service s_3 maintains a data channel used by NF_1 which only processes flows associated with HTTP and HTTPS. This might include, for example, NF_1 including web server-specific intrusion prevention logic.

Each Byway identifies the VM, VMM, and NFs associated with the packet processing *path*. Each of these is associated with the Byway's data channel which is a set of memory regions used for packet communication, and for NF processing of packets. These separate memory regions ensure that NFs only access packets they are explicitly granted access to by their Byway, and gives strong memory isolation between different NFs and different VMs. While utilizing isolated memory regions requires some amount of packet copying, our implementation minimizes the number of copies and optimizes how they are done.

Byways make an important guarantee: All packets that are added into a data channel (either from the VM, or from the network) *must* be associated with the proper `ip:port` pairs, and likewise for packets sent by the NF to the VM or network. This important property ensures that NFs access only permitted flows, and impact only the intended services and network clients. While this means that NFs *are* trusted to process on a service's flows, they cannot impact a VM's network traffic beyond that.

3.2 Byway **Access Control**

A Byway provides access control restrictions to further limit the operations that a NF can perform on its flows. This decreases the impact a NF can have on its service and service clients. It is an effective application of the PoLP to tailor NF access rights to the required rights for it to achieve its goals. These rights center around:

- *A NF's ability to receive data from the VM (*rx-vm*) and/or the network (*rx-net*).* Not all NFs require bi-directional access to packets. A firewall might only need to process on and potentially drop packets from the network, thus does not need to receive from the VM. Thus receive access to packets in either direction can be removed to prevent confidentiality or availability issues for that direction of data-flow.
- *A NF's ability to transmit packets to VM (*tx-vm*) and/or to network (*tx-net*).* Similarly, transmission of packets can be restricted, separately, to the VM or network. This can prevent data leaks and DoS attacks by restricting a NF from transmitting to the net, or by limiting it to transmit only to specific addresses (A in tx-net(A)). A connection logging NF might be unable to modify packets (see the next bullets), and restricted to transmit only to a specific log host and disallowed from transmitting to the VM.
- *A NF's ability to transform/modify packets (*xform*).* Byways support access rights that allow a NF to transform packets (*i.e.,* modify header/payload) either enroute to the VM (xform-vm) or the network (xform-net). NFs implementing an Intrusion Detection or Prevention Systems (IDS/IPS) (*e.g.,* `snort`) do not need to modify packets, thus lack xform. Without xform, a NF cannot corrupt or maliciously modify a Byway's packets.
- *A NF's ability to filter packets (*filter*).* The filter-vm and filter-net access rights allow the NF to drop packets before they reach the VM or network. A firewall often requires this right, while a connection logger should not have it. If a function lacks both xform and filter rights, then it will only be able to observe its Byway's packets, thus guaranteeing their delivery (thus availability) independent of NF logic.

- *The ability to change VM status (*vm-status*).* Some NFs are able to alter the VM's status in response to packet processing, for example to reboot the associated VM after detecting malicious behavior within it.

Each of these access dimensions are specified for individual Byways. A security-centric discussion of these access rights is in §3.5.

Case studies in NF PoLP. We provide examples of how these access rights could be used to enforce the Principle of Least Privilege for sample NFs. In the following, we leave off -vm or -net if both are required.

Firewall (rx-net, tx-vm, filter-vm) Typical incoming firewall functionality for a Byway to permit or drop packets, without ability to update packet contents or see outgoing traffic.

Traffic Encryption (rx, tx, xform) Transparently encrypt all traffic. Note that since the NF is isolated from the VM, the encryption keys are inaccessible to it.

Virtual Networking (rx, tx-net(V), tx-vm, xform) Virtual networking requires encapsulating packets with additional headers for the VM's virtual network range, V. The Byways access rights can enforce that packets leaving the NF fall within the range.

Traffic Monitoring (tx, rx) Classify or analyze traffic flowing in/out of the VM, but without the access right to transform or filter traffic, guaranteeing that the VM's traffic is untouched.

Intrusion-Detection System (IDS) (rx, tx, vm-status) IDS monitors traffic and determines it is indicative of a compromise, in which case it can reboot/recover the VM.

Intrusion-Prevention System (IPS) (rx, tx, filter-vm) IPS monitors network traffic, and if it detects a potential compromise, it will drop the packet(s) before they reach the service.

3.3 NF Isolation and Execution Environment

NFs run in isolated memory protection domains (provided by hardware page-tables), thus only have access to their restricted subset of system memory. NFs have "share nothing" access to memory (excepting their Byway's data channel), with, by default, only the ability to expand their heap. This is paired with a lack of access to any system APIs beyond those involving heap allocation, and thread management (*e.g.,* awaiting the next packet). This inherently limits the scope of erroneous or malicious computation's impact.

NFs execute as threads that are preemptively scheduled to ensure timely progress of NFs and VMs, separately. To maximize performance, existing high-performance NF systems have focused on synchronous ("run-to-completion") NF execution via function calls [4, 16, 48, 49], but such an approach means that faulty or malicious NF logic can delay other NFs or VMs, preventing performance isolation.

This combination of share nothing memory access, limited accessible system API, and preemptively scheduled execution provide strong by-default isolation. The aim is that a NF is only able to impact its services in the associated Byway, and is limited in its ability to do that through the Byway's access control (§3.2).

3.4 VM Execution Environment

To provide strong isolation for traditional (non-NF) tenants, we use VMs. While containers are a popular developer tool (and are often used *inside* of VMs), the inevitability of kernel compromises that enable container escapes has led to most cloud infrastructures using VMs for inter-tenant isolation [1, 2]. Byway Virtual Machines are designed around the ideas of Nova [66] and Firecracker [1]: (1) the Virtual Machine Monitor (VMM) should be instantiated per-VM, thus not constitute a cross-VM attack surface, and (2) the VMM is implemented at user-level in a separate protection domain and uses a thin interface to interact with hardware virtualization acceleration logic in the kernel.

Per-VM VMMs mean that each VMM needs to track only the Byways for its VM. The VMM is key in maintaining Byway access control (§3.2): it implements the packet checking logic to ensure that the `address:port` pairs of packets match the Byway, and match the Byway's access control rights.

3.5 Security Analysis

The design of the Byway abstraction provides a strong, share-nothing NF isolation model by default. This is augmented with access to Byway data channels, and includes dimensions of data channel access control that enable NFs to accomplish their goals, while minimizing their impact on the associated service. We assume that NFs might be malicious or potentially compromised, thus all of a NF's accessible resources (*e.g.,* memory) and APIs comprise the system attack surface. The share nothing, PoLP-centric Byway design directly addresses the CVEs in §2.1 categorized as *privilege escalation*. While a bug in the BywayOS mechanisms could subvert these protections, we minimize the chance of this by achieving a minimal TCB with simple, strong security primitives (capability-based security, as we'll discuss in §4.1).

DoS attacks (§2.1) based on allocating too much memory are trivially addressed by per-NF quotas and restricted means of allocating memory (through a single system API function). Similarly, packet rates are limited with quotas defined relative to the incoming packet rate. CPU processing attacks – where a NF attempts to overuse processing time – afflict run-to-completion runtimes where NF runtimes delay other processing. In contrast, we study how the preemptive NF execution in Byways, paired with progress-preserving scheduling, prevents these issues in §5.4.

Finally, NFs are given access to Byway data channels which, by design, enables them to directly impact their corresponding service. Byway's design mitigates the impact of faulty or malicious NFs by (1) ensuring inter-Byway isolation so that a faulty NF functionally impacts only the service(s) it is associated with, and (2) subsetting the access rights for the Byway to those required to meet the NF's goals. The latter is used to ensure service flow confidentiality (no tx-net), integrity (no xmit), and/or availability (no xform nor filter). While this would not prevent all CVEs, it does enable tenants to make the appropriate trade-offs – assigning specific rights – for their goals. We evaluate (in §5.3) an IDS NF and an IPS NF with only the ability to monitor and assess traffic, a firewall NF with only the ability to drop incoming packets, and an encryption NF with all permissions except the ability to filter packets. Each is used in a Byway including only a HTTP service.

The Byway design does not address a number of additional ways that a NF or a VM could unduly interfere with other tenants. We do not address covert channels, nor attacks through side-channels – for example, through the cache [25, 69], or scheduling [8, 53]. Byway's design is *complementary* to corresponding solutions [25, 29, 41, 45] that are enabled by preemptive NF execution, and use of hardware protection domains.

4 BywayOS **Implementation**

BywayOS is built as a set of components on the Composite μ-kernel [77], which is open-source and publicly accessible [71]. Composite is a component-based system, in which components are similar to lightweight processes – implemented using page-table-based memory protection, and capability-tables [13, 62] – with capability-restricted communication (IPC) connections. System policies are defined in user-level components that interact through optimized IPC as motivated by L4 [37]. Similar to Eros [62] and seL4 [13], Composite uses a capability-based access control model that we leverage directly for Byway security. The Composite kernel is small (around 7k lines of code), includes mechanisms while relying on user-level components to define system policies (such as scheduling [21, 51] and networking), and it focuses on strong multi-core scalability by using no locking while synchronizing using only wait-free means [77].

In contrast to L4-style μ-kernels, Composite uses *thread migration*-based IPC [5, 20, 50], thus defines all synchronization and scheduling policies at user-level [51, 52]. With thread migration, when a client component uses IPC to invoke a function in a server component, there is no thread switch (though all memory context, including execution stacks, is switched), thus execution is accounted to the same thread, and IPC is tracked in the kernel, in per-thread invocation stacks. IPC to invoke a server function (involving cross-page-table coordination) is only allowed if the client owns a *per-function capability* to enable the invocation. We leverage this in

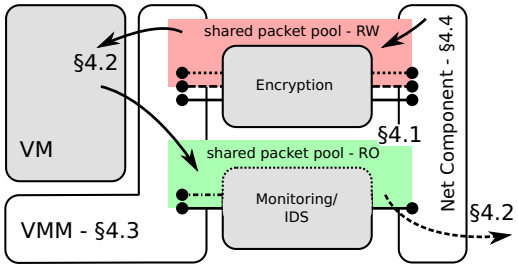


Figure 3: Two NFs with varying access rights including packet pool read or read/write, and the •– variants denote different capabilities to functions to perform specific operations (*i.e.,* receive) discussed in §4.1. In this example, the traffic monitoring/IDS NF has •– capabilities that enable packet reception and the ability to update VM status, but it has no capabilities to modify packets or send them. The encryption NF has capabilities to receive, transmit, *and* transform/modify packets. The data-plane including memory copies (solid arrow) and DMA (dashed arrow) is in §4.2. The virtualization and network components are detailed in §4.3 and §4.4.

BywayOS to ensure strictly constrained service access control, and that all packet copy operations are performed in Byway-specific threads, thus accounting the overhead to the Byway. Figure 3 depicts the core components of BywayOS, all implemented as *user-level* components in Composite.

4.1 NF isolation and Access Control

NF access to packets is controlled using carefully exposed (and capability-limited) system APIs. A NF must have a *capability* to IPC to a specific function in the VMM and network component, thus disallowing access to packets operations, by default. Each of the access rights described in §3.2 correspond to a set of such capabilities.

- Each of vm-status, tx, and rx correspond directly to capabilities to call functions in the VMM and network component to adjust VM status, or send/receive packets.
- The ability to change packets (xform) is implemented as a capability to a function that maps in the shared memory packet pool with read-write permissions in the page-tables. NFs without the ability to change packets can only access a function that provides read-only mappings.
- A NF is able to filter packets if it can deallocate them before they are passed onward. This is enabled by providing access to shared memory reference counters (similar to those in FBufs [12]), or capabilities to functions to modify them.

Importantly, *all* access to operations on packets in a Byway is gated through capabilities. When a Byway is configured to operate on a service for a VM, the appropriate rights are chosen, and capabilities are statically allocated to the NF. Figure 3 depicts these various capabilities (as •– variants) and access rights (read-write (RW) and read-only (RO) packet memory).

§3.2 describes multiple ways to configure these access rights for various NFs. While the ability to receive, transform,

and transmit packets (*e.g.,* required by encryption and virtual networking), or update VM status (*e.g.,* required by the IDS) is straightforward, we will additionally detail access rights for guaranteed packet delivery. For example, the IDS and traffic monitor NFs observe packets, but otherwise cannot interrupt the flow of unmodified packets to and from the VM.

Guaranteed packet delivery requires that a NF does not have capabilities to transmit packets (to avoid double delivery of packets and replay attacks), that it cannot modify packets as it does not have capabilities to read-write packet memory, that it cannot filter packets, and that the system both send packet data redundantly to the NF's Byway, and to the unmanaged Byway. We optimize this path to avoid copying the packet separately into each Byway. Instead, we enable a single packet to safely exist in multiple Byways using reference counts (not directly available to the NF), thus only deallocating the packet when it is no longer referenced by either Byway.

Scheduling. An important part of NF access control is limiting its ability to over-consume core processing time. BywayOS uses a preemptive scheduling policy customized to limit the impact of overly-active NFs on VMs. We use simple round-robin scheduling when NFs complete their computations before the end of a quantum. However, if they become CPU-bound, and might threaten VM throughput, they are scheduled using proportional share (one time unit, for every five of the VM) until they complete execution. We add this policy to the user-level scheduling component of Composite. While simple, this effectively protects VM's CPU shares. More complex policies could be used to minimize tail latency [31, 33], provide proportional progress [67], strict execution bounds [80], or provide strong latency guarantees [61] alongside rate limits [74].

4.2 BywayOS **Data-Plane**

Figure 4 depicts the control and data flow of BywayOS. While there are three Byways, we focus on S_0's blue Byway. BywayOS uses DPDK effectively as a device driver library to receive packets. The Byway-specific code in the networking component then demultiplexes the flow to its corresponding Byway (R1). NF_0 is activated, and invokes (via protected IPC) the network component (via the circular "capability" that enables rx-net), and copies the packet(s) into Byway's data channel memory (R2). The NF processes the pending packets using preemptive, separately scheduled CPU time (denoted by the clock). To transmit the packet to the service, the NF performs protected IPC through the transmit capability (enabling tx-vm) which triggers execution in the VMM (R3). The VMM validates that the transmitted packet has appropriate headers for its Byway, and copies it directly (via `virtio-net`) into the VM's memory (R4).

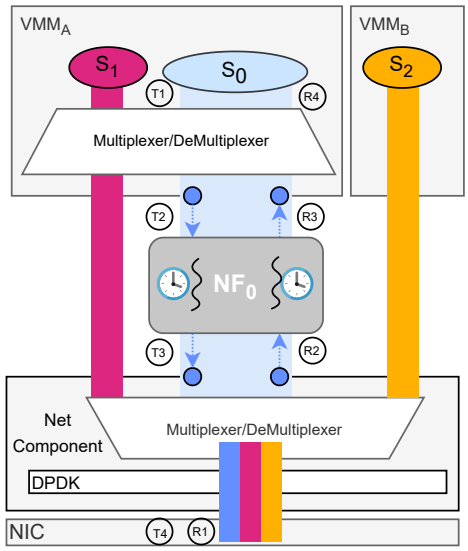


Figure 4: The steps that packets take along the receive (R1-R4) and transmit paths (T1-T4) in BywayOS**. To simplify the image, we omit depicting the VM that the services execute inside.**

On the transmit side, a service sends packets via its OS (recall, the OS and VM aren't depicted here), which transmits the packets using `virtio-net` and traps into the VMM (T1). The BywayOS VMM demultiplexes packets sent from the VM into its Byway, and activates the corresponding NF. When the NF's thread executes, it makes a protected IPC call (using the capability associated with rx-vm) into the VMM, and copies the service's packet(s) into its Byway's data channel (T2). The NF thread returns from the IPC to the NF's protection domain, processes on the packets (using the NF's CPU allocation). When done, to transmit the packet, it performs protected IPC to the network component if it has the corresponding capability for tx-net (T3). The network component leverages DPDK to send the packet directly from Byway data channel memory (after carefully avoiding attacks – see later in this section) (T4).

Both of transmit and receive traverse all of the net component, NF, VMM, and VM, yet packet reception requires two copies, and transmission only once. Protected IPC is an optimized path in the underlying μ-kernel and takes only around 0.3μsec. Frequent thread switches and coordination are mitigated with batching. The rest of this section elaborates on the implementation with details.

Goals. Key goals of the Byways data-plane implementation are to: (1) minimize the costs of packet copying, and decrease the number of necessary copies, (2) ensure that per-packet overheads (*e.g.,* copying) are properly accounted to the Byway's execution, and (3) mitigate the overhead of running many NFs and VMs via time-bounded batching.

Per-packet overheads and accounting. A core security requirement of Byways is that the memory regions used for

each VM, NF, and the network component are disjoint. This requires that packets are *copied* into these isolated memory pools (in (R2), (R4), and (T2)). Thus, to ensure these copies proceed as efficiently as possible, We measure multiple `memcpy` implementations, and selected an implementation based on vector registers and instructions. While kernel code avoids using floating point and vector instructions (to prevent the expensive save and restore operations), our user-level logic is not limited in this way and uses vector registers as normal.

A second optimization we perform is that the same slots in memory are re-used by default when allocating packets into Byway memory pools. This optimizes for the case that the cache-lines for recently processed packets are as close to L1 cache as possible.

Though we aim for high efficiency, copying memory still imposes overheads. Where packet copies must be done, we ensure they are performed using the CPU processing resources of the corresponding NF. Thus, our aim in the network component is to quickly demultiplex [70] packets into Byway-specific packets to identify the NF thread. While we don't use DPDK for this, we do use efficient hash-tables to demultiplex packets into their specific Byways, and wait-free queues to track the Byway's packets. Thus, upon packet reception ((R1)), the NF thread is quickly identified, and the packet is enqueued (though not yet copied).

Only when the NF's thread is executed, does it IPC to the network component, dequeue the packet, and copy it into the pool, making it available for processing within the NF ((R2)). Note that thread migration-based IPC enables our goals: when a NF thread uses IPC to invoke the "receive" function in the network component, the thread safely executes logic in the network component and copies the packets – thus all copy costs are charged to the appropriate NF thread, thus mitigating DoS flood attacks. Only when the IPC returns back to the NF is the thread's access restricted again to that of the NF protection domain.

Eliding packet copies. A straightforward implementation of packet reception from the network might copy the packet (upon networking component demultiplexing) into a Byway memory pool, from the memory pool into VMM memory, and then (through `virtio`'s virtual direct memory access (DMA)) into the VM. Similar for transmission from the VM: copy from VM into the VMM, then into the memory pool for the NF to process, then into the network component. These paths are similar to those in NFQUEUE. To maintain strong isolation between VM and NF (and to strongly isolate the network component and VMM from the tenant's code), we don't avoid all copies, but we do reduce the number where possible as outlined above.

To avoid copying packets redundantly within the VMM, we enable the VMM to directly copy between Byway memory pool and VM (as shown by the solid arrows in Figure 3 and in (T2) in Figure 4). As the VMM has direct access to the Byway's memory pools used by the NF, it transfers packets directly between VM-memory as directed by `virtio-net`, and the memory pool. While this entails a copy, it avoids doing so through temporary bounce-buffers in the VMM. While this is conceptually straightforward, it risks a subtle Time-Of-Check to Time-Of-Use (TOCTOU) [78] attack. These copies can proceed concurrently with VM execution, which means that between when the VMM *checks* packet header information to identify the proper Byway, and when it is *copied*, the VM (or NF) could maliciously update the headers making them no longer appropriate for that Byway. To avoid this, the packet *headers* must be validated and carefully transferred (not directly copied) to ensure that they adhere to the Byway-specific access control.

As outlined above, we also optimize the network component's transmission of NF packets. As the network component has access to Byway memory pools, it is able to DMA the packet *directly* from the NF memory pool ((T4)), completely avoiding the copy (as shown by the dashed arrow in Figure 3). While this avoids a copy, it is also susceptible to TOCTOU attacks. Unlike the VMM that can carefully transfer packets *while* validating packet header contents, the NIC cannot check that a packet being DMAed conforms to a Byway's IP/port. To solve this, the network component can copy the headers into its isolated memory, but not the packet contents. It splits the DMA up into two separate transactions using NIC gather-DMA. This enables the network component to validate that the packet is properly addressed from the Byway, while zero-copying the packet data from the NF. While causing multiple PCI transactions can cause significant overhead [58], we leave finding a break-even point between copying packets and using multiple PCI transactions as future work.

Time-bounded batching and scheduling. NFs and VMs can share the same core, and compete for computing resources. As the NF is on the critical path for network processing, it rarely makes sense to preempt it with interrupts for additional network packets. Similarly, if the VM is busy processing network communication, there is similarly little benefit for preempting it to enqueue more packets. Further, the more that Byways can *batch* process packets at each level (in the network component, NF, VMM, and VM), the less the per-packet overhead. Thus we take an aggressive stance and do not use network interrupts. Instead, we use frequent timer interrupts – every 200μs – to ensure preemptive execution, along with yielding of the CPU (in the VMM on VM idle, and in the NF and network component between batches) to avoid polling overheads.

This naturally pipelines the processing of packets in the various stages of Byway processing enabling high performance. Importantly, it also effectively bounds delays

in network processing. While the timer delay can be tuned, or could be dynamic, we found that the 200μs design point results in a strong throughput and latency (see §5.2).

VMM↔network Byway. The unmanaged Byway for each VM handles all services that aren't explicitly associated with another Byway, avoiding any NF processing. When the VM transmits packets via the unmanaged Byway, they are still copied (by VMM) into shared memory pools for the Byway, and transmitted by the network component (and similar for packet reception). The unmanaged Byway could be optimized to avoid these copies by copying (DMAing) directly out of (in to) VM buffers by integrating the network component's logic with the `virtio-net` logic in the VMM. Instead we err toward simplicity by not intertwining liveness and notification mechanisms of two already complex components.

4.3 Virtual Machine Support

We extend Composite to enable hardware virtualization support through Intel's VT-x [76], and implement a VMM to oversee and control VM execution (shown in Figure 3). A user-level VMM component interacts with kernel support for the hardware acceleration. To support Byways, the VMM includes: (1) `virtio-net` support to transmit and receive packets using the virtual networking interface commonly supported by OSes, (2) Byway routing for packets transmitted from the VM which associates a service with a queue of packets to transmit and the semaphore used to activate the NF thread, and (3) VM APIC/timer and idle traps that enable yielding from the VM to other NFs/VMs.

4.4 Network Driver Component

We use DPDK [32] running in the user-level network component as the driver to communicate with the NIC. We attempt to minimize modifications to core DPDK code, treat it as a naive device driver, and build Byway-specific logic and demultiplexing on top. DPDK is layered in BywayOS so that it could be replaced with verified [55] or simpler drivers [14]. We focus on the *early demultiplexing* of packets to Byways [70] to immediately confine all processing of those packets to their Byways.

The network component includes a separate thread for receiving packets. After DPDK receive queues are emptied, it yields to other (VM and/or NF) execution, thus effectively sharing the core. The network component's path for receiving from NIC queues is optimized for only that purpose. Given the narrow objectives of the NIC receiving logic, we execute it only on a single core. Similar to the VMM, the network component's receive path associates each packet with a Byway, enqueues it into a Byway queue, and activates the associated (VMM/NF) thread.

As discussed in §4.2, the copying of packets is performed by the NF/VMM threads when they IPC to receive packets. The NF/VMM threads that use IPC to transmit packets directly program transmit NIC queues on the thread's core.

5 BywayOS Evaluation

Experimental setup. The testbed of all the evaluations is two dual-socket Dell R740 servers with Intel(R) Xeon(R) Platinum 8160 CPUs running at 2.10GHz with 128GB RAM. Hyper-threading and turbo-boosting are disabled and we use only one socket for experiments to reduce result variability. The two systems are directly connected by two Intel E810 100G NICs without going through a switch. One of the systems is used as the server side running BywayOS or vanilla Ubuntu server 20.04 LTS, the other one which also runs the same Ubuntu server 20.04 LTS is used as the client to generate load. All the virtual machines in either BywayOS or Ubuntu host run the same Linux kernel 5.15.107 with a busybox user level environment equipped with memcached version 1.6.15, and nginx version 1.19. We use QEMU emulator version 4.2.1, VPP 22.02, nDPI 4.6, and the Linux kernel NIC ICE driver version 1.7.16.

Each VM is allocated one vCPU and 300MB RAM. The client benchmarking tools, we use are: the closed-loop memtier_benchmark [44] for all the memcached results; we use two client threads for each VM with the other settings being default (including a data size of 32B, and a get:set ratio 10:1); and the version of wrk2 [79] that allows open-loop testing [23] for the nginx http server results. BywayOS uses DPDK v21.11 with necessary modifications in order to run it in BywayOS. We only rely on DPDK's functionality for packet reception and transmission, thus most of the library is not leveraged.

The systems we compare are:

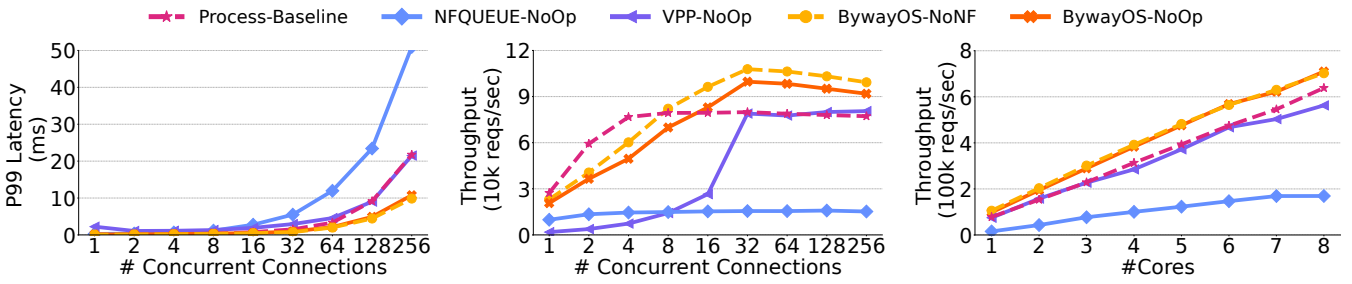
Bare-metal Linux (labeled "process-baseline") which executes the services (memcached or nginx) directly as a process. This is a baseline with *no NF* and *no multi-tenancy*.

NFQUEUE enables NFs to execute in processes, interposed on traffic to VMs. We study both *NoOp NF* in which we use only a NF with pass-through (no-operation) logic to assess system overhead, and practical NFs. We use `libnetfilter-queue1` 1.0.5.

VPP which enables NFs to have direct access to both the network (through DPDK), and the VM's memory. NFs execute as function calls from in the VPP runtime process – we use SR-IOV to enable a separate VPP runtime per VM. We use VPP 22.02 due to incompatibility between newer versions and our NIC.

BywayOS with both *no NF (NoNF)* Byways and with per-service *NoOp* NF Byways. These enable strong and configurable isolation between VMs and between NFs.

We also evaluated various other systems, but found they had worse performance and isolation properties:



(a) Latency (b) Throughput (c) Scalability

Figure 5: memcached Performance

Containers. We found that containers set up to use host networking (*i.e.,* where all services share a port namespace) perform slightly lower than Linux processes (within 5%, in general), and that containers using host bridging (requiring packet encapsulation [85]) express performance between 60% - 70% of Linux processes. Given the weaker isolation properties compared to VMs due to the shared kernel, we omit these results. Note that VPP is a common plugin in Kubernetes' [36] CNI [9] layer, for example through Calico [6]. While we don't compare against container infrastructures, we do compare against key systems that underlie them and to bare-metal Linux that out-performs them.

VMs. We evaluate VMs *without NFs* using `virtio-net` [75] for network communication (both `virtio-host` and `virtio-user`). These approaches could enable QEMU-based NFs, but they achieve less than 40% the throughput of Linux processes. We don't evaluate these further as VPP provides network performance that is stronger than both `virtio` approaches, *while providing an infrastructure for NF execution.*

eBPF. XDP [30] hooks for eBPF programs enabling processing on packets as soon as the networking device driver is done with them. As eBPF programs require enhanced trust (§2.2) (*i.e.,* essentially `root`) and don't support legacy code, we don't focus on them. A conventional means of safely processing on packets in user-level is to use eBPF to pass packets via queues (via `netlink` [46]) to user-level programs (*i.e.,* NFs) for processing. As these approaches are structurally identical to, and have comparable performance to NFQUEUE, we evaluate it.

Service mesh infrastructures. We also evaluated service mesh technologies including Envoy [15]. We found their performance to be consistently less than NFQUEUE. As they share many of the same isolation properties, we restrict our evaluation to NFQUEUE.

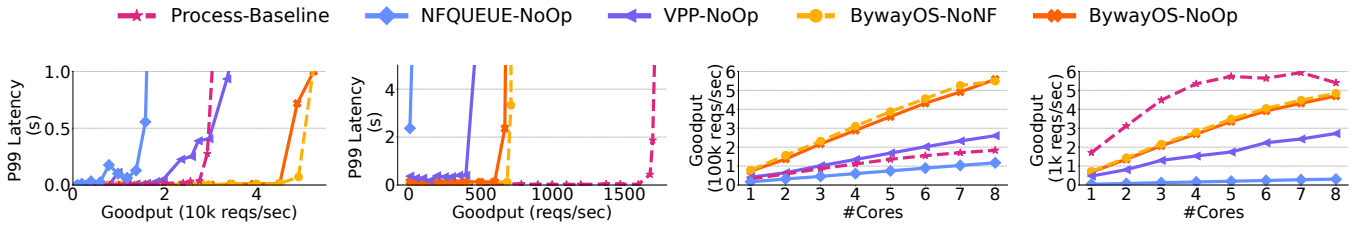
5.1 Performance

To study the network service performance of the systems, we measure throughput and latency separately for memcached and nginx. We focus first on a single core, with all services (NFs, VMs, and system services) sharing a single core. Then we will consider scalability of the systems on multiple cores with multiple VMs and NFs. We evaluate all of the systems, and include both settings that use NFs, and those that don't provide NFs – namely, the process-baseline, and a direct network Byway. All systems that use NFs use *NoOp* NFs that simply pass the packet onward without processing. As such, these results focus on *the system overheads for the different approaches.*

memcached performance. Figure 5 (a) and (b) depict the memcached results for throughput and 99th percentile latency with an increasing concurrency of requests. At their highest throughput, the bare-metal Linux memcached process can perform up to 79.9K TPS with close to 1.5ms p99 latency; similarly, the VPP system achieves comparable throughput with 2.9ms p99 latency; NFQUEUE has significant overhead, and can only achieve 15.5K TPS and a 5.5ms p99 latency; and BywayOS with a NoOp NF achieves 99.7K TPS with 0.93ms, and around 107.8K TPS with 0.75ms with no NF. At lower concurrency levels (less than eight concurrent connections), the BywayOS systems have lower throughput than Linux processes, likely due to differences in batching. Similarly, at concurrency of 32, VPP drastically increases in throughput.

Figure 5(c) shows the scalability of the system with an increasing number of cores. We run one VM and NF per core, thus also scale the tenants. Each tenant VM gets requests from its own client workload generator with concurrency 64. At eight cores, the Linux process approach achieves 638K TPS with a 3.43ms p99 latency, VPP achieves 563K TPS with 11.64ms p99 latency, NFQUEUE achieves 169K TPS with 13.58ms p99 latency, and the BywayOS approaches achieve 709K TPS with 2.32ms p99 latency.

Discussion. Of the approaches that include NF processing, BywayOS maintains the highest throughput, with the lowest p99 latency at all concurrency levels. Even compared against bare-metal Linux processes with no NFs, the throughput of BywayOS is competitive for lower concurrency levels, and stronger for higher levels. This is quite surprising: both systems are running full Linux networking stacks in the VMs, that should have similar performance, but BywayOS also has the overheads of the VMM, NF, and DPDK. This is due to the periodic polling and more efficient batched network processing component that avoids the overhead of frequent



(a) 1KB Page Size **(b) 1MB Page Size** **(c) 1KB Page Size** **(d) 1MB Page Size**

Figure 6: nginx Performance: Goodput and Latency (a, b), Scalability (c, d)

interrupts seen in the Linux baseline. This also results in more packets processed per-virtual interrupt in the VM.

Both BywayOS and VPP see a higher increase in throughput with larger concurrency compared to the process baseline. This is because they both take advantage of batching, which performs better with larger sets of active packets for processing. Byways still performs strongly at lower concurrencies due to its strictly time-bounded batching.

NFQUEUE has low throughput as the frequent Linux IPC to the NF through the host's networking stack imposes significant overhead – both in control costs, and data-copying. However, NFQUEUE does provide the VM with much stronger isolation from NFs than VPP, so this gives a proxy for the costs of NF isolation in existing systems. Overall, BywayOS provides the strongest isolation, with 6.4x and 1.25x the throughput of NFQUEUE and VPP on a single core, and reduces tail latency by 68% to 83%.

nginx performance. We evaluate a web service provided by nginx to understand how the systems react under open-loop workloads (*i.e.,* where requests can outpace systems' ability to reply), and varying payload sizes. Figure 6a and Figure 6b depict the p99 latency and goodput response for each system with an increasing workload request rate with a tenant executing on a single core while serving a 1KB and 1MB webpage. As the workload increases, each system maintains low latency until the system saturates, and the latency "spikes".

For 1KB page sizes, the host Linux nginx HTTP server saturates at 32.2K TPS. NFQUEUE and VPP with NoOp NF achieve a goodput of 16.7K TPS and 37.3K TPS, respectively. BywayOS is able to achieve a goodput of 49K and 48.7K TPS with no NF and with a NoOp NF, respectively. At lower request rates where the systems have spare processing capacity (6K req/sec), BywayOS with NoOp NF has p99 latency of 0.31ms, compared to 24.29ms for NFQUEUE and 3.56ms for VPP.

Similarly, for 1MB page sizes, the Linux process system saturates at 1709 TPS, while NFQUEUE and VPP achieve 41 TPS and 473 TPS. BywayOS achieves 717 and 683 TPS, with no NF and a NoOp NF, respectively. At lower request rates where the systems have spare processing capacity (10 req/sec), BywayOS with NoOp NF has p99 latency of 132.35ms, VPP has 368.89ms, and NFQUEUE has 2370ms.

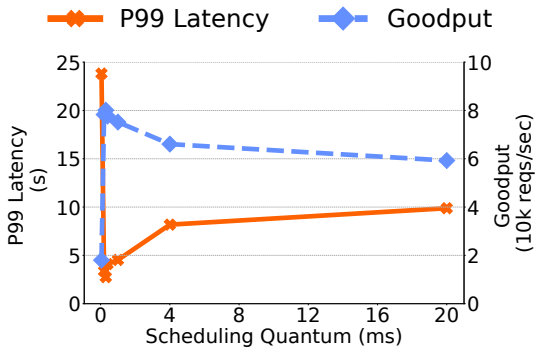


Figure 7: nginx (1KB Page Size) performance under various scheduling quantum

Figure 6c and Figure 6d show the maximum throughput of each system with an increasing number of cores and tenants for 1KB and 1MB webpages. For 1KB page sizes, all systems scale linearly. At eight cores, BywayOS with NoOp NFs achieve 558K goodput while VPP and NFQUEUE achieve 260K and 117K, respectively. All systems scale linearly for 1MB payloads with the exception of the Linux process approach which levels off at five cores. At eight cores, BywayOS with NoOp NF achieves 4.7K requests, which is slightly lower than the host Linux at 5.4K, while NFQUEUE and VPP achieve 0.3K and 2.7K.

Discussion. For 1KB webpages, BywayOS is able to achieve significantly higher goodput than the other approaches – with NoOP NF performance increasing by 1.3x and 2.9x compared to VPP and NFQUEUE, and even outperforming the process baseline (with no NF) by 1.5x. The open-loop workload generates high concurrencies that benefit from our system's time-limited batching. Similarly VPP's batching enables it to outperform the Linux process approach in some cases. However, BywayOS also provides low latency–reducing p99 latency at low load by 78x and 11x compared to NFQUEUE and VPP. For 1MB payloads, the additional packet copies of all approaches that support NFs slow them down relative to the Linux process baseline. Of the approaches that support NFs, BywayOS demonstrates more than 16.6x and 1.4x higher goodput on a single core than NFQUEUE and VPP. Further, at eight cores (past when the Linux baseline hits a scaling bottleneck), BywayOS with a NoOp NF demonstrates goodput within 13% of Linux with no NF.

Page Size	Goodput (reqs/sec)		Performance Degradation (%)
	Optimized Copying	Emulated NFQUEUE Copying	
1KB	75,611	63,258	16%
1MB	667	479	28%

Table 4: nginx performance under varying memory copy strategies.

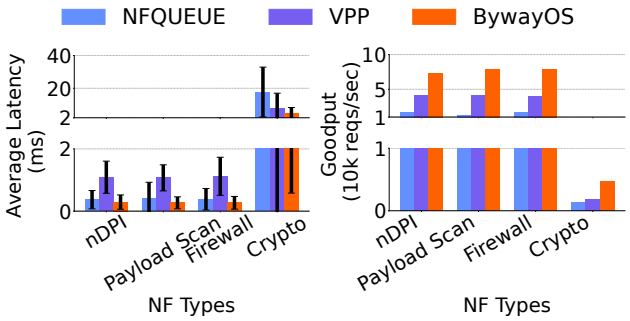
5.2 BywayOS Performance Properties

In this section, we study factors contributing to the BywayOS performance. To more easily reason about the performance properties in BywayOS, we keep the evaluation setup simple and share a single core between a NF and VM. We use the same nginx web server setup as §5.1 with a *NoOp NF*.

Time-bounded batching and scheduling impact. We first evaluate how the time-bounded scheduling impacts performance in BywayOS. Figure 7 shows the nginx p99 latency and goodput serving 1KB web pages under a saturated open-loop workload with an increasing scheduling quantum value. Small quantum values more closely emulate the overheads of frequent interrupts, while larger values decrease context switch overheads, but can impair low-latency communication. When the quantum is significantly smaller than 300μs, the performance degrades – with large p99 latency and low throughput. For example, at 50μs, nginx maintains a goodput of 18K TPS and a corresponding p99 latency of 23.82s, while at 300μs, nginx achieves a goodput of 80K TPS and 2.75s p99 latency, outperforming the 50μs setup by 4.4x in terms of goodput and by 8.6x in terms of p99 latency. With quantum larger than 300μs, nginx performance gradually decreases. At 20000μs, the system achieves a goodput of 59K TPS and 9.88s p99 latency, which is a decrease of 26% in goodput, and 259% for p99 latency from 300μs. We use a quantum of 200μ as we find it provides strong performance across all applications and scenarios.

Data copies optimization impact. We then study how BywayOS's data-copy optimizations impact system performance. Table 4 compares the nginx goodput with a saturating workload between the default setting of BywayOS with optimized data copying, and with 3 copies on the receiving path and 3 copies on the transmitting path which emulates the NFQUEUE data copying overhead. For 1KB web pages, the additional copies cause a 16% overhead, while for 1MB web pages, the overhead increases to 28%.

Discussion. Both the time-bounded scheduling optimizations and eliding packet copies contribute to the performance gains in BywayOS. Context switching that is too frequent (*i.e.,* with too small a quantum) causes significant overheads. In contrast, with a large quantum, performance decreases due to longer TCP handshakes and ack latencies, but is comparably less impacted due to yielding within the VMM, NF, and network components when there is no work to be done. The evaluation also shows the necessity of eliding data copies



(a) nginx Latency **(b) nginx Goodput**

Figure 8: NF Overheads

which cause both direct costs due to the copy operation, and indirect costs as caches are increasingly used for copy buffers. Even for workloads with small packet sizes, extra data copies still cause a 16% slowdown, which increases to 28% for larger packets.

5.3 Network Function Execution

We evaluate the systems that support executing NFs with various network functions that vary in terms of required permissions and per-packet execution overheads. We focus on existing, legacy NFs and port the following to VPP, NFQUEUE, and BywayOS. We port the following from Open NetVM (ONVM) [82]: (1) a string matching payload scan to detect and prevent specific attacks (`payload_scan` in ONVM). (2) firewall processing to control port and IP access (`firewall` in ONVM), (3) AES encryption and decryption of traffic (`aes_decript`/`aes_encrypt` in ONVM), and (4) nDPI [10] (version 4.6), a heavyweight library for deep packet inspection. Figure 8 depicts nginx workloads with these NFs interposed on the service processing path. Figure 8a depicts the latency of HTTP requests at a request rate of 1K/second (before system saturation), and Figure 8b shows the maximum goodput for each NF.

Discussion. Consistent with previous results, NFQUEUE achieves less throughput than the competing approaches. However, its latency is often comparable to, or less than, VPP. This is due to VPP's batching that leads to higher throughput, but increased packet latency as they are handled in chunks, delaying driver interactions with the NIC. BywayOS provides significantly higher throughput and lower latency for all NFs.

5.4 VM and NF Isolation Properties

In this section, we assess the impact of malicious or faulty NFs on the VMs for which they process packets. Many attacks are avoided *by design* due to NFs executing in a separate protection domain, with separate data channels, and restricted access control. However, as discussed in §3.5, DoS attacks (§2.1) – especially those on the CPU – require dynamic scheduling policies constrain their impact, thus we focus on their

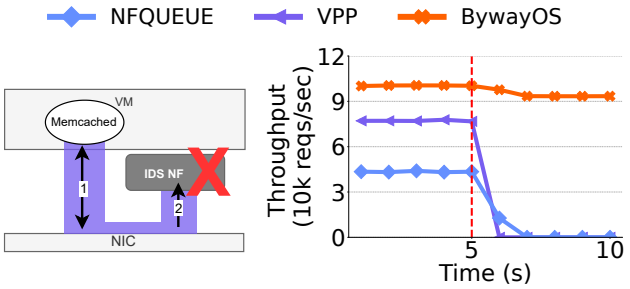


Figure 9: Guaranteed packet delivery to `memcached` service. The Byway continues to deliver packets despite NF failure via path 1.

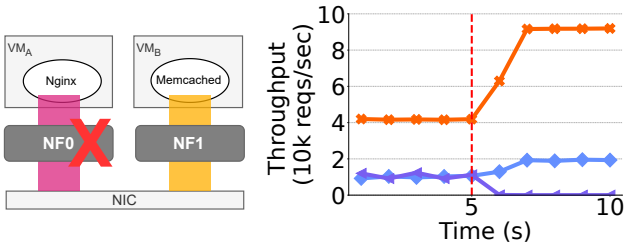


Figure 10: `memcached` throughput when there is a failure in `nginx`'s NF. With inter-Byway isolation, `memcached` and its NF maintain throughput.

evaluation. Figure 9 and Figure 10 report the throughput for memcached over time. They show a timeline with a system executing normally up until the fifth second. At that point the NF triggers a malicious behavior by going into an infinite loop.

Figure 9 shows a system with a single memcached Byway with a service with *guaranteed packet delivery* (*i.e.,* for an IDS). As such, when the NF fails, BywayOS maintains high throughput, while VPP and NFQUEUE suffer complete service degradation. Even with a failing NF in BywayOS, due to the inability of the NF to modify packets and the guaranteed delivery policy, the service carries on. The slight service degradation experienced is due to processing contention on the core, but the service maintains high throughput due to BywayOS's preemptive, proportional-share scheduling.

Figure 10 demonstrates that even for NFs that require the ability to modify and filter packets, services on separate Byways maintain strong isolation. Here, a Byway for nginx has a NF that fails, and we only report the throughput for a separate memcached Byway. As VPP NFs execute within the same context as DPDK, this prevents all future network processing. NFQUEUE, on the other hand enables separating out services using `iptables`, so it maintains service for memcached, actually increasing throughput for memcached as nginx no longer contends for CPU time. BywayOS ensures memcached service continuity, similarly increasing its throughput as nginx core contention is removed. While both NFQUEUE and BywayOS maintain memcached availability, Byways do so with over 4x the throughput.

	VM + NFQUEUE	VM + VPP	BywayOS
VMM	QEMU: 5.13M		BywayOS VMM: 9K
	Firecracker: 51.7K		
Network Device	Linux Ice NIC: 106K	DPDK: 133K	DPDK*: 122K BywayOS: 1.3K
Kernel/ System Services	Linux Kernel: 2.77M		Kernel*: 7.8K
			Scheduler*: 1.9K
Virtualization Driver	KVM: 45K		BywayOS driver: 0.9K
Total (LoC)	2.97M	2.99M	142.9K (*20.9K*)

Table 5: The Trusted Computing Base (TCB) of each system for NF execution. We report Lines of Code (LoC) as measured with `tokei`. The VMMs for NFQUEUE and VPP can be *either* QEMU or Firecracker, so both lines are included, but the total assumes Firecracker. Total LoC for BywayOS is "with DPDK (*without DPDK*)".

5.5 Complexity and Trusted Computing Base (TCB)

Table 5 depicts the various software packages that are part of the TCB of each NF system. NFQUEUE and VPP depend on the VMM (QEMU [57] or Firecracker [17]), the kernel KVM driver, and the rest of the Linux kernel. VPP additionally depends on DPDK for direct NIC access. The table reports the Lines of Code (LoC) of each package as reported by `tokei`.

For DPDK, we exclude drivers for other devices, libraries not used, example code and test code. For the Linux kernel, we focus on code in the x86-64 architecture, and exclude most device code, drivers, and file systems not used on our system. For QEMU, we focus on x86-64 architecture code and exclude other platform code and test code. Recall that for VPP, the NF itself has complete access to the VM as well (§5.4), so the NF is part of the VM's TCB.

For BywayOS, software that we use mostly unchanged is marked with $*$: DPDK is minimally modified (reported on the BywayOS line), the core Composite kernel is unchanged, and the scheduler is modified to add preemptive, proportionate scheduling. For DPDK, we remove the virtual networking (virtio and vhost) code, as it isn't used in BywayOS. All other reported lines are Byway-specific code.

Discussion. While Lines of Code are not a perfect metric for measuring software, they do convey some information about software complexity and attack surface. BywayOS reduces the total lines of code in the trusted computing base by a factor of 20x compared to either VPP or NFQUEUE. Not only is the TCB for providing VM and NF isolation small in BywayOS, it is built on the Composite μ-kernel, which further isolates various aspects of the system from each other (*i.e.,* the scheduler and DPDK are in separate protection domains in user-level), and BywayOS carefully uses capabilities to tightly focus NF access on only the desired access rights.

Vigor [81] found that only 3.5% of DPDK's code is typically executed for common NFs, and only 18% of an Intel NIC driver. This implies that the actual TCB is effectively smaller for

simple uses of DPDK, or could be much smaller for verified NICs. We could significantly shrink BywayOS TCB by using the `ixy` driver which is less than 1K LoC [14], or verified networking device drivers [55]. Table 5 includes BywayOS's TCB without DPDK, 20.9K LoC, to serve as a basis for this, a 142X reduction in TCB compared to VPP/NFQUEUE, while offering stronger isolation properties.

6 BywayOS Generality and Limitations

Generality of Byways**.** While our contributions in this paper mainly focus on: (1) introducing the Byways abstraction for running untrusted NFs in a multi-tenant environment on the cloud, and (2) providing a existence-proof (the BywayOS) of the Byways abstraction, demonstrating that it can both have *strong isolation properties* and *high performance*, we believe it is also possible to adapt Byways into existing systems.

Most of the Byways' design and implementation techniques are applicable in other systems. For example, its *intelligent scheduling* can be either added into the Linux kernel or integrated with newer user-level scheduling extension architectures like the ghOSt [31]; the *copying optimizations* are possible by optimizing the packet processing path of the VMM (*e.g.,* QEMU) and the Linux kernel; the Byway-*aware network drivers and VMMs* could be added into a fast path of the Linux kernel (*e.g.,* with XDP [30]) and integrated into existing VMMs; the Byway-*specific constrained packet access rights* might be implemented with shared memory regions and appropriate kernel extensions to provide appropriate access control; and the *constrained NF execution environment through capabilities* can be emulated using system-call limiting techniques with SELinux [40] extensions or seccomp [60].

Some other abstractions are not easy to emulate. BywayOS's accounting of copies to NFs, and fast IPC are challenging to replace in existing monolithic systems. However, batching might mitigate the impacts of expensive IPC. Certainly, a small TCB can only be achieved with a smaller system such as BywayOS.

In short, we believe that many of Byways abstractions can be added into existing systems, leveraging existing mechanisms, thus generalizing the Byways beyond BywayOS.

Limitations and future work. The current BywayOS implementation supports only one NF per Byway. For more complex NF scenarios that require complex NF switching, chaining and composition, BywayOS can be extended with complementary solutions. First, BywayOS' switching mechanisms in the network component and VMMs is simple, but can be expanded to scalable and complex logic as in OVS [54] and VFP [18]. Second, compositions of various NFs can be compiled together into a *single* BywayOS *NF component* – for example, connected via switching logic from OVS [54], VFP [18], or via programmatic composition as in Click [35, 43]. This solution does not isolate NFs within composition from each other, which is common in most NF infrastructures. This enables compositions of logical NFs to be implemented as a Byway NF, while still providing strong isolation between the NF compositions. Third, we can extend Byways to include chains of isolated NFs as separate BywayOS components. To maintain strong isolation, this requires additional inter-NF copies that will impact performance. However, EdgeOS [59] and EdgeRT [61] demonstrate that chains of isolated NFs rely on copying while still achieving higher than expected throughput. We believe that BywayOS is a strong foundation to be extended into these more general directions.

7 Conclusions

This paper presents Byways, a new abstraction for multi-tenant hosts to safely execute NFs efficiently. Byways bind a NF to a set of services, and prevent the NF from impacting other services, even in the same VM. Byways are configured with access privileges such that their NFs are only be able to perform operations on traffic that are appropriate for the NF's functionality. This tightly limits how NFs can impact their VMs and services, constraining the negative side-effects of errant or malicious NFs.

We implement Byways in BywayOS and demonstrate a 20X reduction in the number of lines of code contained within its trusted computing base compared to existing approaches. Despite this strong isolation, Byways achieve high performance. A memcached service in a Byway VM can achieve throughputs and p99 latencies often better than memcached executing in Linux *without virtualization*, and more than 4x higher throughput than NFQUEUE that comes closest to providing strong NF isolation. For small webpages served with nginx, Byways achieves nearly 3x higher throughput than NFQUEUE, and is 30% faster than VPP, which offers little fault isolation. For eight VM/NF pairs running large webpages that stress copying overheads, Byways can achieve throughput within 13% of a native system with no network functions, and 1.7x to 15x better than NFs running in VPP and NFQUEUE.

Byways demonstrate that even in a VM infrastructure, per-service paths serve as a useful first-class abstraction for isolation and resource management. BywayOS demonstrates that strong isolation does not have to come at the cost of performance and can be mitigated with specialized and targeted aggressive optimization. Correspondingly, we believe that Byways can serve as the foundation for a new cloud infrastructure enabled by multi-tenant NFs.

Acknowledgements. We'd like to thank our shepherd and reviewers for their time and effort that significantly improved this paper, and Samy Bahra for his generous support. This work is supported by NSF CPS 1837382 and ONR N000142212084. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI*.

[2] Amazon Web Services. 2024. The Security Design of the AWS Nitro System - AWS Whitepaper. *AWS Whitepaper* (Feb. 2024). https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf

[3] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling End-Host Network Functions. *ACM SIGCOMM Computer Communication Review* 45, 4 (Aug. 2015), 493–507. https://doi.org/10.1145/2829988.2787493

[4] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. 2018. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine* 56, 12 (Dec. 2018), 97–103. https://doi.org/10.1109/MCOM.2018.1800069 Conference Name: IEEE Communications Magazine.

[5] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, and C. Small. 1999. Pebble: A component-based operating system for embedded applications. In *Proc. USENIX Workshop on Embedded Systems*. 55–65.

[6] Calico: Cloud Native CNI Implementation, https://github.com/projectcalico/.

[7] Hyunseok Chang, Murali Kodialam, T. V. Lakshman, Sarit Mukherjee, Jacobus Van der Merwe, and Zirak Zaheer. 2023. MAGNet: Machine Learning Guided Application-Aware Networking for Data Centers. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 291–307. https://doi.org/10.1109/TCC.2021.3087447

[8] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. 2019. A Novel Side-Channel in Real-Time Schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[9] Container Network Interface (CNI): https://www.cni.dev/.

[10] Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano. 2014. nDPI: Open-source high-speed deep packet inspection. *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)* (2014), 617–622. https://api.semanticscholar.org/CorpusID:383106

[11] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. 2003. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[12] Peter Druschel and Larry L. Peterson. 1993. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Symposium on Operating Systems Principles*. 189–202.

[13] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 what have we learnt in 20 years of L4 microkernels?. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 133–150.

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

[15] envoy. https://www.envoyproxy.io/.

[16] extended Berkeley Packet Filter (eBPF). https://ebpf.io/.

[17] Firecracker 2019. Firecracker: https://firecracker-microvm.github.io/.

[18] Daniel Firestone. 2017. VFP: a virtual switch platform for host sdn in the public cloud. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 315–328.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[20] Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*.

[21] Phani Kishore Gadepalli, Runyu Pan, and Gabriel Parmer. 2020. Slite: OS Support for Near Zero-Cost, Configurable Scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 160–173.

[22] Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, and Zach Day. 2019. Chaos: a System for Criticality-Aware, Multi-core Coordination. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[24] Rahil Gandotra and Levi Perigo. 2020. NFEH: An SDN Framework for Containerized Network Function-enabled End Hosts. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 1–6. https://doi.org/10.1109/ICCCN49398.2020.9209701

[25] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys '19)*.

[26] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, Peter Pietzuch, and Rüdiger Kapitza. 2018. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 386–397. https://doi.org/10.1109/DSN.2018.00048

[27] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 681–693. https://doi.org/10.1145/3387514.3405895

[28] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[29] Amir H. Hashemi, David R. Kaeli, and Brad Calder. 1997. Efficient Procedure Mapping Using Cache Line Coloring. In *Proceedings of the*

ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation.

[30] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. 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*. 54–66.

[31] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[32] Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[34] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

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

[36] Kubernetes: https://kubernetes.io.

[37] J. Liedtke. 1995. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM.

[38] Will Lin, Yizhou Shan, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. 2024. SuperNIC: An FPGA-Based, Cloud-Oriented SmartNIC. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 130–141. https://doi.org/10.1145/3626202.3637564

[39] Linux KVM: http://www.linux-kvm.org, retrieved 9/16/12.

[40] Peter Loscocco and Stephen Smalley. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 29–42.

[41] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time Cache Management Framework for Multi-core Architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[42] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

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

[44] memtier_benchmark: https://github.com/RedisLabs/memtier_benchmark.

[45] Mitra Nasri, Thidapat (Tam) Chantem, Gedare Bloom, and Ryan M. Gerdes. 2019. On the Pitfalls and Vulnerabilities of Schedule Randomization against Schedule-Based Attacks. In *Proceedings of the Twenty Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '19)*.

[46] Netlink: `netlink` 7 man page.

[47] NFQUEUE: https://netfilter.org/projects/libnetfilter_queue/doxygen/html/index.html.

[48] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*.

[49] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[50] Gabriel Parmer. 2010. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*.

[51] Gabriel Parmer and Richard West. 2008. Predictable Interrupt Management and Scheduling in the Composite Component-based System. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*.

[52] Gabriel Parmer and Richard West. 2011. HiRes: A System for Predictable Hierarchical Resource Management. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[53] Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh B. Bobba. 2015. A generalized model for preventing information leakage in hard real-time systems. *2015 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2015).

[54] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[55] Solal Pirelli and George Candea. 2020. A simpler and faster NIC driver model for network functions. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[56] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar

[57] QEMU. https://www.qemu.org/.

[58] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

[59] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. 2020. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.

[60] Seccomp BPF. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[61] Wenyuan Shao, Bite Ye, Huachuan Wang, Gabriel Parmer, and Yuxin Ren. 2022. Edge-RT: OS Support for Controlled Latency in the Multi-Tenant, Real-Time Edge. In *IEEE Real-Time Systems Symposium (RTSS)*.

[62] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Symposium on Operating Systems Principles*. 170–185. citeseer.ist.psu.edu/shapiro99eros.html

[63] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. 2012. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2342356.2342359

[64] snort. https://www.snort.org/.
[65] SR-IOV: PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. 321211-002, Revision 2.5.
[66] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems* (Paris, France) *(EuroSys '10)*. 209–222.
[67] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. 1996. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Real-Time Systems Symposium*. IEEE.
[68] strongswan. https://strongswan.org/.
[69] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. 2011. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 38th annual international symposium on Computer architecture* (San Jose, California, USA) *(ISCA '11)*. 283–294.
[70] David Tennenhouse. 1989. Layered Multiplexing Considered Harmful. In *Protocols for High-Speed Networks*. North Holland, Amsterdam, 143–148.
[71] The Composite Component-Based System source: https://github.com/gparmer/composite.
[72] Kashyap Thimmaraju, Saad Hermak, Gábor Rétvári, and Stefan Schmid. 2019. MTS: bringing multi-tenancy to virtual networking. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 521–536.
[73] Kashyap Thimmaraju, Gábor Rétvári, and Stefan Schmid. 2018. Virtual Network Isolation: Are We There Yet?. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges* (Budapest, Hungary) *(SecSoN '18)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3229616.3229618
[74] Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. 2018. Tableau: a high-throughput and predictable VM scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference*.
[75] Virtual I/O Device (VIRTIO) Version 1.3, Committee Specification Draft 01, Referenced from 06 October 2023, https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html.
[76] VTx: Intel Systems Programming Guide, Volume 3C, chapter 24, Order Number: 325384-083US, Updated March 2024.
[77] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. 2015. Speck: A Kernel for Scalable Predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[78] Robert N. M. Watson. 2007. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *First USENIX Workshop on Offensive Technologies (WOOT 07)*.
[79] wrk2. https://github.com/delimitrou/DeathStarBench/tree/master/wrk2.
[80] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. 2011. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT '11)*.
[81] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.
[82] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. http://faculty.cs.gwu.edu/timwood/papers/16-HotMiddlebox-onvm.pdf
[83] Yang Zhou, Mark Wilkening, James Mickens, and Minlan Yu. 2024. SmartNIC Security Isolation in the Cloud with S-NIC. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 851–869. https://doi.org/10.1145/3627703.3650071
[84] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. 2023. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3620678.3624652
[85] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.