Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds

Yuxin Ren, *The George Washington University;* Guyue Liu, *Carnegie Mellon University;* Vlad Nitu, *INSA Lyon France;* Wenyuan Shao, Riley Kennedy, Gabriel Parmer, and Timothy Wood, *The George Washington University;* Alain Tchana, *ENS Lyon France*

https://www.usenix.org/conference/atc20/presentation/ren

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds

Yuxin Ren[1], *Guyue Liu*[2], *Vlad Nitu*[3], *Wenyuan Shao*[1], *Riley Kennedy*[1], *Gabriel Parmer*[1], *Timothy Wood*[1], *Alain Tchana*[4]

[1] *The George Washington University* [2] *Carnegie Mellon University*
[3] *INSA Lyon France* [4] *ENS Lyon France*

Abstract

5G edge clouds promise a pervasive computational infrastructure a short network hop away, enabling a new breed of smart devices that respond in real-time to their physical surroundings. Unfortunately, today's operating system designs fail to meet the goals of scalable isolation, dense multi-tenancy, and high performance needed for such applications.

In this paper we introduce EdgeOS that emphasizes system-wide isolation as fine-grained as per-client. We propose a novel memory movement accelerator architecture that employs data copying to enforce strong isolation without performance penalties. To support scalable isolation, we introduce a new protection domain implementation that offers lightweight isolation, fast startup and low latency even under high churn. We implement EdgeOS in a microkernel based OS and demonstrate running high scale network middleboxes using the Click software router and endpoint applications such as memcached, a TLS proxy, and neural network inference. We reduce startup latency by 170X compared to Linux processes, and improve latency by three orders of magnitude when running 300 to 1000 edge-cloud memcached instances on one server.

1 Introduction

The Internet of Things foretells the deployment of billions of devices requiring processing close to the data source to avoid excess bandwidth consumption in the network core. Similarly, latency sensitive cyber physical systems desire communication and processing at millisecond scale, preventing the use of standard cloud platforms. Use cases such as these motivate the demand for "edge clouds", tiny data centers that can be deployed as close to users as possible (*e.g.* at an Internet Service Provider (ISP) or a nearby telco central office [55]).

An edge cloud site is expected to serve a large number of clients with high performance. Many edge services such as Network Function Virtualization (NFV) middleboxes that must act as a "bump in the wire" are latency-sensitive and throughput-intensive. However, given the large number of edge cloud sites, each is expected to only have a small number of powerful servers due to space, power, and cost constraints (*e.g.* the HPE EL4000 has 64 cores and AWS Snowball Edge has up to 52 cores). To utilize resources in an efficient, elastic and scalable way, an edge cloud must support dense multi-tenancy—each edge cloud will be *highly resource constrained* compared to a centralized cloud, yet it may need to host many *securely isolated* services for the clients connected to it, and often these clients have a short lifespan (*e.g.* a mobile user), leading to high churn.

Unfortunately, the combination of limited resources, large number of clients, and diverse services of edge clouds pose major challenges for traditional system software designs. To protect clients and services, an edge system needs to provide two types of isolation:

- **Client Isolation**: Multiplexing an edge service among multiple clients exposes them to malicious exploitation that could impact every client (*e.g.* a compromise in the TLS implementation as in Heartbleed). Thus ideally, untrusted clients should not share a protection domain (*e.g.* a process, a container, or a virtual machine).
- **Service Isolation**: An edge server needs to serve multiple services from different tenants. Some services may be vulnerable and tenants may even be malicious. Thus, a service should not share any resources, such as memory, with other untrusted services.

Current systems fail to provide both high performance and strong isolation–particularly between clients. Recent Network Function Virtualization platforms achieve high throughput with the use of kernel-bypass networking and zero-copy techniques, but they often trade isolation for performance [23, 27, 72]. This works for a single service, but the edge cloud needs to serve multiple services from different tenants. Lightweight virtualization techniques based on unikernels [32] and hypervisor optimizations [47] have been proposed to reduce boot times and density, but don't address providing many isolated clients high throughput. Recent support for HW virtualization, such as SR-IOV capable NICs, reduces virtualization layer costs, but comes at the expense of

scalability. It works for a few dozen clients, but can't be used for an edge server that needs to support thousands of clients.

We address these challenges by designing EdgeOS, a new system that achieves the difficult combination of strong isolation, efficient communication, and fast boot times. Our key idea is to dynamically start *a new isolated domain for each client*, and to use *data-copying* to move messages. This idea is based on two intuitions. First, for a large number of short-lived clients, starting a new protection domain can be more efficient and secure than maintaining many long running yet infrequently accessed ones. Second, in contrast to long-standing networking subsystem guidance that dictates that zero-copy is necessary [24, 66] – often at the price of isolation, we observe memory can push data at sufficiently high rates for edge environments such as 5G base stations that have bandwidths in the low 10s of Gb/s [21, 39]. Thus memory copying can provide stronger isolation, without becoming a performance bottleneck as long as it is faster than line-rate.

Based on these insights, EdgeOS contributes:

- A carefully optimized "Memory Movement Accelerator" (MMA) communication and buffer management architecture that enforces isolation with data copying, while retaining high throughput and low latency.
- A "Feather Weight Process" (FWP) that redefines the process abstraction to a minimal memory footprint and set of capabilities needed to support dense deployments of edge computation, and provides strong isolation between each client in multi-tenant environments.
- A control plane with flexible routing and FWP chain caching to support microsecond speed initialization of complex services in high churn environments.

Combined, these features produce a novel architecture that eschews the current trend towards zero-copy I/O in order to provide stronger *per-client* isolation, yet still offers better performance scalability, reduced tail latency, and dramatically better support for high churn edge environments than any system we are aware of.

We extend the Composite μ-kernel [67] to implement the EdgeOS prototype. We target two key categories of edge applications: network functions (e.g., middleboxes from the Click software router [28]) and latency sensitive endpoint services (e.g., HTTPS servers, neural network inference, and the memcached key-value store). These services can be combined to build flexible service chains, while providing stronger isolation and latency guarantees than existing approaches.

Our evaluation illustrates how our isolation and communication abstractions offer dramatically better scale, density, and performance predictability than traditional approaches. We execute 1000s of FWPs per host, instantiate them 170X faster than a Linux process, maintain a `memcached` latency under 1 ms even when running 600 isolated instances on a single host, improve the throughput of HTTPS processing by almost a factor of 2.3, and even CPU-bound neural network inference tail latency improves by almost 50%.

2 Motivation

We first introduce our threat model and isolation properties. Then we discuss performance challenges that edge clouds pose to existing isolation platforms, motivating the need for a redesign of the underlying communication mechanisms and OS primitives.

2.1 Threat Model

There are three types of parties in our model: (1) A system run by the trusted edge cloud operator that provides isolation mechanism and hosts edge services. (2) Edge services deployed by different cloud tenants who supply untrusted code or binaries. (3) Untrusted clients who send requests to the edge services of one or more tenants. The goal of attackers is to compromise security systems, exfiltrate user data, or disrupt edge services. We assume an attacker has capabilities to evade system security mechanisms by exploiting vulnerabilities in the edge service binaries. We consider two general attacker cases: malicious tenants and malicious clients.

Malicious tenant. A tenant could provide malicious or vulnerable services in order to affect the operation of services run by other tenants. After initialization, a tenant's services are trusted only with the permissions given to them by the system for specific resources, such as memory, communication endpoints or system calls. However, a service can make arbitrary use of the permitted resources regardless of whether they are shared by other services.

Malicious client. A client could try to tamper with other clients' traffic by exploiting a vulnerable service. Clients can request any service or send arbitrary packets. After a client successfully attacks a service, we assume it can access any data or resources that are permitted to the controlled service.

EdgeOS seeks to grant resource access permissions to services and enforce isolation among them in order to limit the effects of malicious tenants and clients. In particular, the system wants to maintain tenant-isolation (i.e., a malicious service should not be able to disrupt services from other tenants) and client-isolation (i.e., a malicious client that exploits an instance of a service should not be able to affect other clients). We do not attempt to prevent a malicious tenant's services from affecting its own clients, just as a normal cloud provider does not try to validate client services.

Isolation model. EdgeOS provides a strong form of isolation based on constraining both inter-tenant *and* inter-client (running code for a specific tenant) interference. Tenants provide *chains* of FWPs, each of which executes as a separate, preemptive thread, and protection domain (including page-table-constrained memory). As such, FWPs access disjoint sets of read/write memory, interact only with adjacent FWPs in their chain using message passing, and receive proportional execution time. The lack of shared resources (*e.g.*, no shared memory) and ambient authority (*e.g.*, no shared filesystem namespace) provide strong logical isolation. Preemptive

scheduling policies prevent CPU-based resource consumption attacks. EdgeOS ensures that FWPs in a chain cannot be bypassed, and that the output packets cannot be modified by upstream FWPs. As such, FWP chains constitute a high-performance implementation of assured pipelines [7].

We enable a *chain* of FWPs to processes client requests – as opposed to requiring a tenant to provide a *single* FWP– for multiple reasons: (1) FWPs at the start and end of the chain can be required by the system and provide the likes of firewalls and rate-limiting, (2) some applications are naturally implemented in a separate address space, thus using multiple FWPs to provide legacy support, and (3) it allows tenants to more strongly isolate at-risk computations from those that are more important (*e.g.* TLS termination).

EdgeOS's strong isolation between FWPs ensures isolation between tenants. When paired with fast FWP instantiation, it provides per-client isolation. New connections addressed to a tenant's service can (optionally) be served by a *separate* FWP chain, thus lifting the inter-FWP isolation to provide both inter-tenant, and inter-client isolation.

2.2 Existing Isolation Options

In order to support extreme dense per-client isolation, we propose that a protection domain should have the following properties. 1) it is *sealed* [26] so both the binary and configuration cannot be modified after initialization; 2) it has minimal access to system APIs and resources; 3) it cannot share any resource, such as memory, with other untrusted protection domains; 4) once a client's computation is finished, instead of reusing its protection domain – which would allow compromises to impact future executions – it is re-initialized to a safe state.

In contrast, current systems that use process pools or virtualization do *not* provide this inter-instantiation isolation. Existing solutions provide weaker isolation:

- UNIX processes are exposed to large system call interface and TCB in the kernel. Even containers using more security features, such as `cgroups`, `namespaces`, `seccomp-bpf` and `chroot`, still maintain significant state (including signals, file descriptors, memory mappings) that increases attack surfaces.
- Virtualization encapsulates a hardware abstraction along with multiple enclosed processes and system state. Thus it introduces an extra hardware-enforced isolation boundary. Though research has optimized implementations [2,32], the memory overhead, and startup latencies are not sufficient for per-client isolation.
- Language techniques use software-based isolation, but either don't provide temporal isolation, instead executing tenant computation non-preemptively [52] or using heavy-weight language runtimes [22].

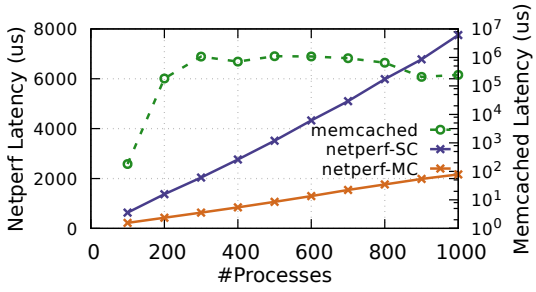

Figure 1: Round-trip latency of N `netperf` or `memcached` instances. Compared with the 1ms round-trip of 5G networks, `netperf` latencies represent a 2x/8x latency increase using one/sixteen cores, while `memcached` exhibits a 1000x latency increase.

2.3 Multi-tenancy and Churn

Given the increasing number of stakeholders that can benefit from edge cloud execution, supporting multi-tenant execution is critical. Network slicing [1, 45, 49] is essential to best utilize edge resources for 5G networking. The challenge [58] is to efficiently share the relatively constrained resources at the edge (often between less than one and low tens of racks [13,14]), while efficiently isolating tenants. Complicating this is the dynamic behavior [43, 44] of these systems which requires adaptation to the environment's inherent churn.

Churn and isolation overheads. Unfortunately, even relatively efficient mechanisms such as containers impose significant overhead when new clients require isolated computation. This is because those mechanisms rely on layers of abstraction and management of a large number of namespaces.

Percentile	Docker	Firecracker	fork()	EdgeOS
50th	521	126	0.26	0.048
90th	574	129	5.8	0.054

The table above depicts the cost in milliseconds of leveraging various isolation facilities; we measure the time to start a minimal service and then fault in 8 pages of memory to show the unpredictability of Linux's Copy on Write (full details in Section 5.2). Using `docker start` can take hundreds of milliseconds due to the cost of initializing namespaces and setting up Docker [10] metadata. Amazon's Firecracker [2, 19] still takes over one hundred milliseconds. Even Linux `fork()`, which has a much lower cost than Docker, exhibits high variance, with the 90^{th} percentile being over 20 times slower than the median. In contrast, our EdgeOS platform improves median start time by 5X compared to Linux, and has minimal variability. Later we show we can improve EdgeOS by another order of magnitude by maintaining a cache of fresh services that can be started near instantaneously.

2.4 Latency and Throughput at Scale

Lightweight isolation mechanisms such as containers facilitate running large numbers of applications (e.g., hundreds of Docker containers per server), but they cannot provide

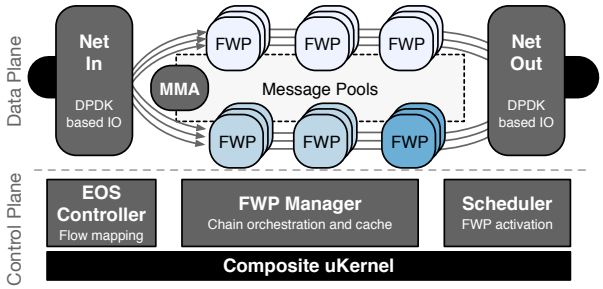

Figure 2: EdgeOS Control and Data Plane Architecture

performance predictability as the scale rises. This leads to the second key challenge in edge infrastructures: predictable performance, particularly latency, at large scale.

Scaling isolation facilities. Unfortunately, current infrastructures suffer poor performance not only under churn, but also at high scale. Both VMs and containers see overheads due to the expense of traversing the host's software switch to determine the appropriate destination to deliver incoming data. Even prevalent and widespread OSes such as Linux suffer from this issue. To evaluate the latency behavior of Linux, we adjust the number of `netperf` servers sharing a single core (`netperf-SC`) or spread across multiple cores (`netperf-MC`), and the number of `memcached` instances spread across multiple cores. A second, well provisioned host transmits traffic to the test server over a 10 Gbps link. Using multiple cores still cannot achieve ideal latency due to poor scalability as shown in Figure 1. Real applications such as memcached are quickly overwhelmed and can only support a hundred or fewer instances (full details in Section 5.6). This illustrates the inability of existing OS isolation mechanisms to provide fine grained performance isolation at high scale. EdgeOS is designed to support isolation with both high scalability and predictability.

3 Design

Figure 2 shows the overall EdgeOS architecture, with trusted components having white lettering. The EdgeOS data plane is composed of Memory Movement Accelerators (MMA) that efficiently and securely copy data between services deployed by tenants as Feather-Weight Processes (FWP), which can be composed into chains to build complex services. The EdgeOS control plane instantiates and schedules these components and routes messages to them.

3.1 Design Principles

Our EdgeOS is designed under the guidance of widely accepted secure system design principles [8, 35, 64].

Avoid shared resources (P1). Every shared resource may open an attack channel [35]. EdgeOS avoids sharing of all types of resources, such as memory, communication endpoints and system services to prevent malicious activities.

Mediated communication (P2). This principle states that communication should be passed via a trusted component, and rules out shared memory based communication. Within EdgeOS, the kernel and MMA mediate all communication initiated from untrusted services.

Least privilege (P3). This well-known principle requires every component to have minimal privileges to limit damage from a system compromise. However, current isolation mechanisms, such as containers or VMs, are usually running on top of monolithic systems, whose kernel or hypervisor has full privilege. EdgeOS applies this principle not only to untrusted components, but also low-level system services, such as MMA and scheduler.

3.2 Memory Movement Accelerators (MMA)

Copy-based communication. Existing high throughput systems [51, 52, 72] often eschew isolation and use shared memory to pass data among isolated services. In contrast, EdgeOS eliminates shared memory between services (**P1**). A key EdgeOS design is that all communication between untrusted services use *data copying* and are mediated by MMA (**P2**).

As long as memory copying is higher bandwidth than the network line-rate, it is a viable form of data movement that provides strong isolation. On our processor, memory throughput is 472 Gb/s, and though networking throughput is ever-increasing in the data-center, it is more limited on the edge. Practically, in §5.1 (Figure 5(c)) we show that the MMA can sustain throughput competitive with a middlebox framework – that avoids copying by sharing packet memory – up to 54 Gb/s. For perspective, 5G cells provide on the order of between 2Gb/s [21] and 20Gb/s [39]. This design is counter to long-standing networking subsystem guidance that dictates that zero-copy is necessary [24, 66] – often at the price of isolation, EdgeOS optimizes the MMA implementation and treats it as a specialized processor (§4.2) to achieve the line-rate. As a result, EdgeOS maintains strong isolation without practically sacrificing performance.

Efficient data copying with the MMA. The MMA acts as a software DMA engine to move message data between services, and runs on one or more *dedicated cores* in order to perform out-of-band data movement. The MMA retrieves messages from an upstream service's message rings, copies them and adds them into a downstream service's message rings, and alerts the scheduler that the destination service needs to be activated to receive it. EdgeOS further separates memory into a *message pool* that is used for communication, and *local memory* for each service's local state. This separation enables memory allocations to be optimized for the purpose and use of the memory (§4.1). MMA only has the access rights to copy into, or from message pools (**P3**).

Network Gateways. In and Out gateways leverage DPDK [11], run on dedicated cores[1] and pull packets into message pools with no kernel interactions. MMA then copies

[1] Note that current edge offerings include between 20-64 cores, and we show (§5) that, in aggregate, EdgeOS is efficient despite specializing cores.

packets into the destination service, thus enabling strong protection among both (untrusted) edge services and (trusted) system services.

3.3 Feather-Weight Processes (FWP)

A Feather-Weight Process (FWP) is a minimal abstraction wrapping only memory and a small set of simple kernel resources. FWP achieves strong isolation by (1) capability-based access control which minimizes access to the rest of the system; (2) library-based services to avoid sharing of sensitive information; (3) FWP caching that re-initializes a FWP's context before serving a new client.

Capability-based resource isolation. In EdgeOS design, access to all resources relies on capability-based access control [9] using kernel-mediated references, removing any ambient authority [37] (**P2, P3**). These resources include local memory, the message pool that is used to receive and send data, and synchronous communication end-points to request operations from system-level services. Capabilities to memory are enforced by hardware page-tables, while other capabilities are protected by the kernel. Each FWP is encapsulated within its own capability space, which restricts the granted resource to only the allowed tenant. No memory is shared between FWPs, instead MMA copies data between FWPs.

Library-based services. Notably absent in EdgeOS are default access to conventional shared OS services. Similar to Unikernels [30, 31, 69], FWPs make use of library-based implementations [18], thus enabling the inclusion of only the application-required services without sharing with other FWPs (**P1**). We have ported a TCP/IP networking stack and a simple in-memory file-system to FWPs. The memory-based file system is used to store transient and configuration data. If global persistent state is required, then network-accessible storage services can be used (similar to a serverless computing model). This decoupling enables a simplified and efficient FWP execution environment to enable high density and line-rate computation.

FWP Caching. A new client should not be allowed to see old context accumulated from previous clients. Current practices either ignore this, such as process pools, or manually terminate and restart the service [3, 5], which repeatedly incurs unnecessary initialization overhead. EdgeOS employs an FWP checkpoint cache, that both avoids reusing possibly compromised state of previous executions and avoids redundant initialization computations. In doing so, EdgeOS guarantee that (1) an FWP is sealed so it cannot be modified after checkpointing; (2) an FWP is restored to the cached post-initialization state. Thus its memory is placed into a known and safe state, ensuring the integrity of future FWP instances. Checkpointing details are described in §4.3.

FWP Chains. To provide more complex functionality, FWPs can be arranged into chains, thus the entire chain can be efficiently managed as a whole. A FWP chain composes multiple checkpointed FWPs and are maintained in a *FWP-chain cache* that caches entire chains of FWPs, their interconnections, and their message pool. MMA copies data between adjacent FWPs within the same chain, avoiding shared memory.

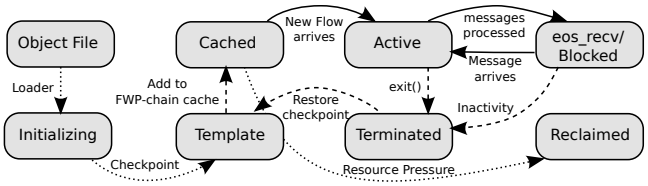

Figure 3: Lifecycle of a FWP-chain: Dotted lines indicate FWP manager operations conducted once to load and then checkpoint a FWP-chain, or to reclaim the FWP's resources when memory pressure exists. Dashed lines indicate operations to re-initialize terminated FWP-chains for future use. Solid lines are *data-path* operations performed by on the critical path.

3.4 EdgeOS Control Plane

The EdgeOS Control Plane is composed of: (1) the EdgeOS Controller that maps incoming flows to FWP chains, (2) the FWP Manager that controls the lifecycle of FWPs and optimizes their startup, and (3) the Scheduler that determines which FWP to run on each core and activates them in response to incoming messages.

Flow matching with the EdgeOS Controller. When new requests arrive from connected client devices, they need to be routed to the appropriate FWP chain. The EdgeOS Controller allows tenants to define FWP chains and the packet filtering rules that specify what traffic should be routed to them. These rules are pushed to the Net-In data plane component. Net-In applies rules similar to SDN match-action rules: packets are split into flows based on the header n-tuple (*e.g.* src/dest IP and port) and a rule is found that matches the flow. The rules indicate the FWP chain that will process that flow.[2] Since our focus is on fine-grained isolation and high scale, a rule can indicate whether all flows that match the rule should be handled by a single chain, or if each client flow should be given a dynamically started instance of the chain.

FWP Manager. Figure 3 illustrates the lifecycle controlled by the FWP Manager. Similar to a Linux process, an FWP starts as an object file, which must be loaded into memory. Once execution begins, FWPs perform some initialization routines, and are checkpointed to a Template. Then multiple identical copies are forked off the Template and put into FWP cache. As new clients arrive, they are paired with corresponding FWP-chains from the cache. The selected FWPs will be Activated, allowing them to process messages or transition to the Blocked state, before eventually Terminating when no longer needed. When a FWP chain terminates, the Manager reuses the chain by Restoring it to the post-initialization state and puts it back into the FWP-chain cache. If there is memory pressure, cached FWP templates and chains are Reclaimed.

[2] Our implementation currently assumes flow rules are statically preconfigured, but this could be extended to support on-demand flow lookups similar to SDN controllers, with a northbound interface to application logic that would assign a rule dynamically to each flow.

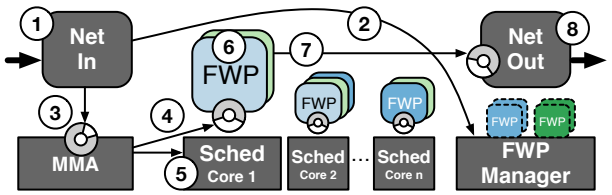

Figure 4: EdgeOS Timeline

Scheduling and inter-FWP coordination. Once a set of FWPs are activated, they are distributed across cores, and partitioned scheduling (*i.e.* without task migrations) multiplexes the core's processing time.

Traditional systems often use shared data-structures and Inter-Processor Interrupts (IPIs) for scheduling notification. For example, Linux activates threads by accessing that thread's data-structure directly to see if it is already awake, and if not, an IPI is sent. The resulting cache-coherency traffic and IPI overheads, can be significant, especially if used for message notifications arriving over a network at line rate. Motivated by these overheads, NFV platforms based on DPDK such as OpenNetVM [72] use active polling for communication between threads on different cores, thus avoiding blocking. However, as the number of processes ("network functions" in OpenNetVM) grows beyond the number of cores, spin-based event notification is inefficient.

All inter-scheduler coordination in EdgeOS is via message passing, avoiding shared memory synchronization. When a FWP-chain is activated, or when a message is sent to an FWP, the MMA notifies the scheduler of the activation. On the other hand, a FWP will be blocked after processing all of its messages. FWPs avoid spinning to ensure efficient multi-tenant computation. We currently use a simple and efficient preemptive, fixed-priority, round-robin scheduling policy. This aims to provide *temporal isolation* between untrusting FWP chains which prevents them from monopolizing the CPU, and from interfering with the progress of other tenants' FWPs.

Timeline Summary. Figure 4 shows the complete timeline for processing a packet. (1) A packet reception at the Net-In gateway causes a flow lookup to decide which FWP chain should process the packet. (2) If there is a miss, the FWP Manager spawns a FWP chain from its cache. (3) MMA copies the packet and (4) adds it to the first FWP's ring of the destination FWP chain. (5) MMA messages the scheduler on the FWP chain's core to activate it. (6) The FWP chain processes the packet and (7) the last FWP in the chain asks the output gateway to DMA the packet out the NIC.

3.5 Isolation Analysis

We summarize how EdgeOS achieves the isolation requirements listed in §1 based on our design principles. EdgeOS executes on top of a micro-kernel, with a smaller TCB than monolithic systems (on the order of 10K lines of code). FWPs access system resources through a capability-based access-control system provided by the kernel [67]. Capabilities protect and control access to kernel resources including synchronous and asynchronous IPC end-points, threads, time [20], and memory. The capability model is similar to that in seL4 [17], and relies on user-level retyping of untyped memory into both kernel resources and virtual memory. The most notable difference between EdgeOS's capability model and seL4's is that EdgeOS doesn't allow IPC-based delegation, instead relying on a user-level component with capability-based access to a FWP's capability-table to copy capabilities (thus access to kernel resources) into that FWP. EdgeOS leverages the kernel's capability system to tightly constrain FWP's access to system resources. Each FWP has access to only its own memory and to IPC endpoints to the scheduler, and to the FWP manager to block awaiting further execution, and expand their heap, respectively.

Trust model: §2.1 discusses the threats from untrusted tenant code, and from clients that can compromise that tenant code. As such, we assume that any resources available to an FWP will be used, where possible, to escalate privilege, and that all FWP system interfaces will be comparably stressed.

EdgeOS is implemented as a set of trusted, user-level components that have access to various FWP resources. The MMA has shared memory access to each FWP's message pool, and is trusted to properly move messages between adjacent FWPs, to and from the network gateways, and to notify schedulers of FWP activation. Similarly, the network gateways (using DPDK) are trusted to properly interface with the NIC and the MMA and to properly implement a tenant's flow matching rules. Per-core schedulers [54] have the ability to dispatch FWP threads, and are trusted to properly preemptively schedule FWPs and coordinate with the MMA to properly activate them. The FWP manager is relied on to quickly and correctly create new FWP instances, perform capability delegations into the FWP, maintain the FWP cache, and dynamically expand FWP heaps. The FWP manager delegates resources to FWP chains such that all readable/writable resources are partitioned per-FWP. Thus, each chain is mutually isolated, and when a chain is assigned to a client, clients are mutually isolated. Finally, the kernel is depended on to maintain the capability-based access model that constrains the set of resources available to each FWP.

FWP isolation: FWP memory is initially allocated by the FWP manager, and is of three types: (1) shared executable and read-only data derived from a tenant's FWP's image, (2) read-write memory in the global data segment and heap that is *not* shared among FWPs, and (3) message pools that are shared only with the MMA. Finally, each FWP has access to IPC end-points to request heap expansion, and to await the next message's arrival. Messages are sent downstream, and received from upstream FWPs using message pool, thus interfacing with the MMA using only simple wait-free ring buffers. The MMA maintains associations between an upstream FWP's message pool and the downstream FWP, thus only allowing data-flow within a chain. EdgeOS's design ensures that a

malicious FWP – or a compromised FWP – will not be able to intercept data outside of the FWP's chain, nor impact the integrity of correct services.

Inter-FWP *temporal isolation* is enforced by preemptive scheduling. Each FWP executes in a separate thread controlled by per-core, round-robin scheduling logic. In contrast, many of the most efficient language-based techniques (e.g., NetBricks [52]) cannot prevent a buggy or malicious tenant's infinite loop from preventing progress for all tenants.

Inter-client isolation: Each client is served by a separate FWP (chain), the FWP (chain) is re-initialized before serving each new client, and when created, each FWP (chain) is delegated disjoint read/write resources by the FWP manager. Thus, a malicious client cannot impact the execution of future clients. In contrast, VM techniques often create a VM per-tenant [2], which executes multiple clients, thus potentially exposing clients to past compromises. Even webservers are typically architected to service multiple clients within a single protection domain.

When inter-client sharing is required by a tenant (*e.g.*, to implement a shared cache), FWP chains can either access network-accessible storage, or use Net-In gateway rules that send new clients to an existing FWP chain storing common state (§3.4). The former is similar to the stateless design of many serverless and microservice applications; in fact, FWPs provide more flexibility since per-user state can be maintained across multiple requests if desired. In the latter case, isolation is traded for sharing, which is evaluated in §5.6. More complex routing rules that cluster specific sets of clients into different, potentially existing, FWP chains are beyond the scope of this paper.

Per-client isolation in EdgeOS is, in many ways, most similar to systems to provide Distributed Information Flow Control (DIFC) [40]. Such systems track information as it flows (via IPC and other interactions) between processes, and define policies to determine when that flow is allowed. When a single process is needed in two conflicting information flows, a common strategy [15, 41, 62, 71] is to create a *new instance* per flow. This is similar in mechanism and motivation to the per-client FWP chain instantiation in EdgeOS. Importantly, EdgeOS focuses on providing instantiation of full FWP chains, low overhead (an order of magnitude less than Linux `fork`), and line-rate performance. Despite strong FWP isolation, EdgeOS achieves *per-packet* overheads on the order of dedicated middlebox infrastructures that do not provide comparable isolation.

4 Implementation

We implemented EdgeOS in Composite (`composite.seas.gwu.edu`), an open source μ-kernel that externalizes traditionally core kernel features into user-level *components* that define the resource management and isolation policies [67]. In Composite, components interact through highly-optimized Inter-Process Communication (IPC) to leverage system logic and resources. Composite is based on a capability-based protection model [17, 61] that controls component access to kernel resources. The kernel includes no scheduling policies, instead implementing schedulers at user-level [54]. The Composite kernel scales well to multiple cores as it has no locks and is designed entirely around store-free common-paths, wait-free data-structures, and quiescence [67]. MMA can be implemented in other OSes such as Linux. In EdgeOS we pair it with the FWP abstraction to provide fine-grained isolation and adaptability to churn.

EdgeOS prototype consists of the MMA, FWP management, DPDK-based network access and schedulers. In total, EdgeOS adds fewer than 6000 lines of code. We plan to release our code and experiment templates for repeatable research.

4.1 Message Pool Management

Memory management integration into ring-buffers. Each FWP's message pool is associated with two ring buffers that track *both* how to transmit and receive messages, *and* the allocation and deallocation of messages. EdgeOS observes that general purpose memory allocation facilities (`malloc/free`) can have significant overhead. Thus, we integrate memory with message management by tracking free memory in rings.

A reception ring buffer contains a set of references to message slots into which incoming data can be copied, and the transmission ring buffer contains references to messages to move downstream in the FWP chain. The MMA orchestrates data movement between different FWP's packet memory regions, thus acting as a software DMA accelerator.

Message pools are managed by FWPs as a span of MTU-sized message slots, and unlike traditional NIC DMA ring buffers, the ring buffers include an entry for *each* message slot. Ring entries that have been transmitted by an FWP, and have been copied by the MMA are marked as free, and are used for packet allocations. FWPs must maintain a sufficient number of messages in reception rings to buffer messages that queue up due to the system's scheduling latencies. Thus, after FWPs finish processing pending messages, they move batches of freed messages from the transmit ring into the reception ring. This avoids `malloc` on the fast path, as message *liveness* is managed indirectly through the ring buffers.

Message pools and isolation. The ring buffer design decouples the *message pool* from the *meta-data* to coordinate the data movement and liveness between FWPs and the MMA. This avoids lock-based protection of the rings, instead relying on wait-free mechanisms. This is necessary to avoid the high costs of synchronization, and ensure progress of the MMA in spite of possibly malicious FWPs.

4.2 Memory Movement Accelerator

Our initial experiments showed that naively copying packets in a DPDK-based NFV pipeline decreased throughput

by more than 50%. However, a MMA core has a throughput of around 54 Gb/s on our hardware, which is sufficient for line-rate. For networks that require a higher throughput, more cores can be specialized as MMAs. In the limit, MMA's throughput is bounded by the chip's memory bandwidth, which for our processor is 472 Gb/s. By using the parallelism of the underlying processor and specializing cores to run the MMA, we achieve both isolation and high throughput by taking message movement out of the critical path.

The MMA has read-write access to all message pools. It maintains a mapping between both pairs of transmit and receive ring buffers for subsequent FWPs in a chain, and continuously iterates through all such pairs, transferring messages when it finds a transmitted message. The MMA provides two essential services: *data-movement by copying transmitted messages*, and *event notification of the receiving FWPs*. The MMA's FWP event notification is efficient as it simply sends a message to the scheduler controlling the target FWP's, and relies on the scheduler to asynchronously process events.

MMA optimizations. As the MMA is on the data-path of all FWP interactions, including message reception, it must be able to move messages at faster than line rate. The data-structures linking transmit and reception rings are laid out in an array to leverage the processor's prefetcher as the MMA iterates over them. The initial implementation of the operations on the ring buffers were straight-forward, but cache-coherency traffic, possibly a cache-line transferred for each ring entry, hurt throughput. To address this, we optimize the MMA:

- Double-cache-line (128B) *caches* are added to both the enqueue and dequeue operations. These caches are in local memory outside of the ring, thus their modifications avoid coherency traffic. When retrieving to the ring, a batch is copied into the cache, and when transmitting messages are queued in the cache, and batch copied into the ring. To ensure message delivery, the cache is flushed by an FWP before it blocks.
- These caches enable messages to be transferred in batches. We use explicit software prefetch instructions to load all referenced messages in the cache to avoid CPU cache misses on message processing.
- Messages are efficiently addressed and copied as the MMA has shared memory access to all message pools. To maintain protection, the MMA validates that FWP messages are within a valid message pool.

4.3 Optimized FWP checkpointing

EdgeOS caches the images of *chains* of FWP binaries so they are ready for prompt activation. These ready-to-execute images are *asynchronously* prepared, thus moving the overhead for FWP preparation off the fast-path. The cached FWP's state is identical to the *initialized* state of a ready-to-execute FWP.

We utilize a few optimizations to efficiently generate post-initialization FWP snapshots: (1) the post-initialization checkpoint of the FWP-chain is laid out contiguously in memory so that chain re-initialization is as close to `memcpy` / `memset` overheads (for which we use the `musl` libc, unoptimized versions), (2) we do not eagerly reclaim – and thus later re-allocate – heap memory from terminated FWPs, instead only zeroing it out to maintain confidentiality, and using it to satisfy future heap allocations, (3) we reuse the threads active in each FWP by only resetting their registers to the appropriate post-initialization state, which avoids the overhead of thread destruction and allocation, and (4) only if there is memory pressure do we reclaim first spare FWP heap memory, then cached FWPs. Re-initialized FWPs maintain zero state from their previous execution: the stack, heap, and writable data sections are reset to the initial state. These optimizations culminate in a system that can handle exceedingly high churn and scalability: FWP chain initialization is dominated by `memcpy/memset` overheads, and new client chain activation takes in the low 10s of μ-seconds.

5 Evaluation

All experiments are run on CloudLab Wisconsin c220g1 series nodes [57]. These are 2 socket, 8 core, Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz processors with 128GB ECC Memory. Note that these systems have fewer cores than current edge offerings, thus pressuring EdgeOS's design that dedicates cores to different functions. Systems are connected via Dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes).

5.1 Latency and Throughput

We first evaluate the latency and performance predictability of EdgeOS compared to other high performance networking platforms. Figure 5(a) shows the response time distribution (in microseconds) for an ICMP ping response Click [28] element implemented as either: a DPDK process, an OpenNetVM NF (ONVM), a standard linux process with kernel-based IO, a ClickOS NF in a Xen VM, or an FWP in EdgeOS. The results show that EdgeOS significantly outperforms all of these techniques (by up to 3.8X in average latency), except for DPDK. DPDK is slightly better because it can run only a single service at a time and thus does not need to copy packets from the initial receive DMA ring to a separate pool. In contrast, EdgeOS provides a platform to potentially run thousands of distinct services, and thus needs to offer stronger isolation via copying.

Figure 5(b) shows the maximum throughput of different approaches when forwarding traffic from `pktgen`, a high speed packet generator. EdgeOS again provides better performance than ClickOS, while offering stronger isolation than DPDK and ONVM, which rely on globally shared memory pools for zero-copy IO.

Next we compare the performance of EdgeOS communication with ONVM. We run a chain of NFs on the same core that each forward small (64B) or big (1024B) packets, thus both

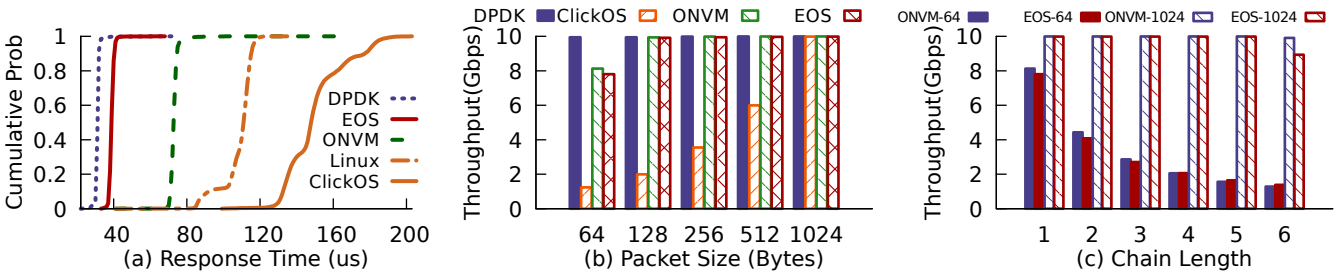

Figure 5: (a) EdgeOS provides substantially better latency, and reduced jitter compared to Linux processes and NFV platforms like OpenNetVM and ClickOS. (b) Throughput of each system with different packets sizes. (c) EdgeOS provides isolation and adds negligible overheads compared to OpenNetVM (no isolation) for different chain length for messages of size 64 and 1024 bytes.

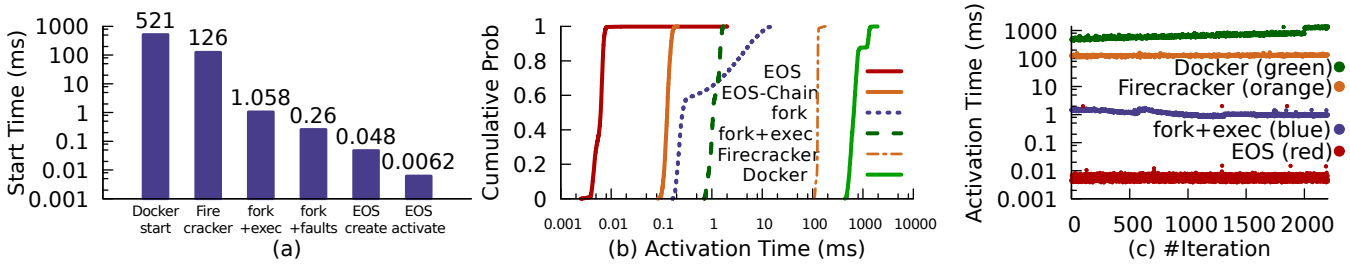

Figure 6: EdgeOS provides orders of magnitude better startup time than other approaches and does not suffer from scalability problems when starting larger numbers of FWPs.

systems have context switch overhead by passing a packet to the next NF. In addition, EdgeOS has copying overhead from the MMA to enforce isolation. The results in Figure 5(c), show that as the chain length increases, the throughput of 64B packet drops for both EdgeOS and ONVM. The main overhead of EdgeOS is data copying, while the overhead of Linux context switches and scheduling dominates ONVM. When the chain length is smaller than 3, the overhead of copying is less than 8%, and EdgeOS outperforms ONVM when the chain is longer as the Linux system overheads increase. The throughput with 1024B packets maintains line rate for both systems when the chain length is smaller than 6, at which point EdgeOS sees a throughput decrease. Even at length 6, the MMA is able to maintain an aggregate of 54 Gb/s.

5.2 Startup Time

FWP Initialization and Activation. In Linux, initializing a process involves calling `fork` (and possibly `execve`). For Docker containers, a `docker run` command is similar, but includes additional system calls to configure namespaces and maintain container metadata. For Firecracker, we use the recommended "hello" image and use 1 vCPU and 128 MiB RAM. In order to optimize the fast path of readying a cached FWP, EdgeOS separates out creation from activation. For EdgeOS, creation involves transitioning from the Object File to Cached state in Figure 3, including setting up page tables, capability tables, and thread creation. We record the start time for 10,000 iterations of starting a container, VM, process, or FWP and report the median in Figure 6 (a). Note the log scale. We use median time values as Container creation cost increases slowly over time so the mean is skewed by these outliers. We compare against two variants of Linux processes: "fork + exec" loads a different binary whereas "fork + faults" mimics loading the service's working set by issuing writes to eight different pages to trigger page faults (the size of the minimal FWP). These approaches are 5-20X slower than the comparable "EOS create" approach (dashed lines in Figure 3).

Once an FWP has been created, EdgeOS keeps copies of it in a cache which can be quickly activated on demand (solid lines in Figure 3). Cached activation improves EdgeOS performance by another order of magnitude, allowing new processing entities to be instantiated in 6.2 microseconds. Figure 6(b) presents a CDF of these approaches, including the activation cost for a full chain of 10 isolated FWPs, which remains an order of magnitude faster than fork+exec.

FWP Scalability and Middlebox Computation. Containers and VMs suffer from poor scalability: as the number of instances rise, the start time increases [32]. In Figure 6(c) we show the time to start a new container, create a Firecracker VM, exec a process, and activate an FWP, when up to 2200 are started incrementally. The Container case gradually drifts upward before hitting a step after 2000 containers (note log-scale) – the last container takes 1.368 seconds versus 0.467 seconds for the first. FWPs provides nearly constant start time regardless of scale. EdgeOS has a few outlier points (11 out of 15K measurements are at 2ms), which we believe to be Non-Maskable Interrupts, or a bug in our scheduling logic.

5.3 Isolation

Just in Time Service Instantiation. To evaluate the impact of client churn in edge environments, we measure client response time for a ping that creates a new FWP. Clients send requests at a configurable interval, and we assume that each new client requires a new, isolated FWP. The new FWP re-

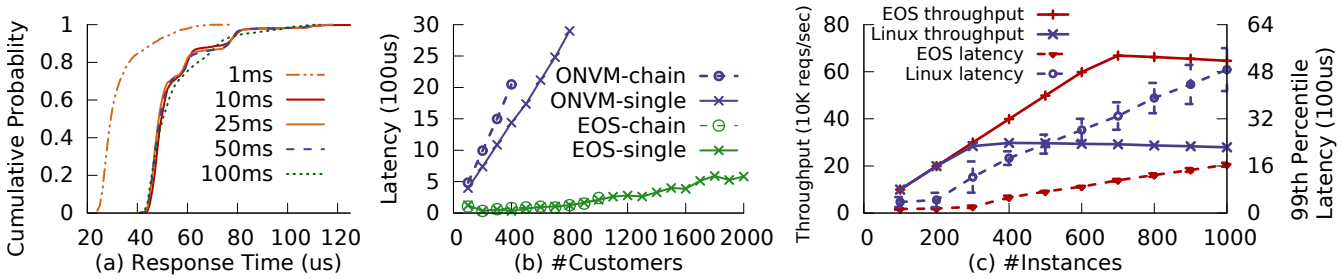

Figure 7: (a) EdgeOS just in time service instantiation for each for mobile client connection with varying client inter-arrival rates; (b) Routing and processing latency for middlebox routing `netperf` traffic for an increasing number of clients; (c) TLS termination performance for up to 1000 end points (solid lines: throughput, dashed lines: tail latency).

ceives the incoming packet, produces a reply, and then terminates, representing a worst case churn scenario. Figure 7(a) shows a response time CDF for EdgeOS under different client arrival patterns. The results show that even when a new client arrives every millisecond, 90% of requests are serviced within 50 microseconds This experiment mimics that in LightVM [32], and although we have not been able to successfully run the LightVM software on our testbed, we note that their paper produced a 90th percentile response time of 20 milliseconds (more than 400X worse) with 10ms client arrivals. The EdgeOS performance advantage comes from our extremely lightweight FWP abstraction and our template cache that allows nearly instant instantiation.

Multi-Tenancy and Customer Isolation. An important job of edge-cloud systems is to act as a middlebox to monitor traffic close to the source. Figure 7(b) depicts the processing latency of a middlebox deployed between `netperf` client and server machines for an increasing number of concurrent clients. We use three nodes, two running `netperf` clients and servers, and the third running EdgeOS or ONVM in the middle. The systems run either a single firewall to filter flows or a 2 FWP chain of firewall plus monitor, all implemented in Click, to further maintain statistics about flows. Each customer is serviced by its own "personal firewall" or chain, thus preventing malicious clients interfering with others. We measure the middlebox latency overhead (i.e., the added cost versus direct client/server connections from Figure 1) as we increase the number of clients, and thus number of FWPs (EdgeOS) and Network Functions (NFs in ONVM).

Though ONVM is a highly optimized middlebox infrastructure, it relies on containers and expensive coordination mechanisms between NFs and the management layer. Because of this, ONVM cannot scale past around 820 containers or 410 chains, and the added latency rises quickly with each new client. FWPs enable the system to scale past 2000 clients with an average increase in the latency of only around 0.3μs per additional client. Chaining in EdgeOS adds negligible latency overhead thanks to efficient FWP scheduling and activations, while ONVM sees an increasing gap since it relies on Linux's more heavyweight futexes.

5.4 TLS Termination

For our first edge cloud use case, we consider the deployment of edge-based TLS termination proxies, such as for a CDN serving `https` traffic or for IoT devices sending encrypted data streams. This requires an edge end-point for TCP connections, a TLS handshake to share public keys, and continued encryption/decryption of transferred contents. As the majority of web traffic is over `https` [59], TLS implementations are a high-priority target for compromises, and have a history of high-impact vulnerabilities, e.g., Heartbleed [12]. Therefore, instead of sharing one `https` end-point among many clients, we instantiate an isolated TLS FWP for each client. Toward this, we ported `axtls` (`axtls.sourceforge.net/`) (version 2.1.4), which includes a lightweight `https`-based web proxy optimized for embedded systems, and the `lwip` (`savannah.nongnu.org/projects/lwip/`) TCP/IP networking stack (version 2.1.2) to EdgeOS.

In our experiment we use a single cloudlab node as the edge server, and use five additional nodes to drive the client workload. We style this experiment after the setup in [32], and request zero-length files hosted at the proxy (headers are still encrypted). Each client uses `ab` (version 2.4.39) and keep-alive sessions to make a series of requests over a TLS-encrypted session. We modified `ab` by adding a `nanosleep` to rate limit each client to 1000 requests per second. We disable Nagle's Algorithm for these experiments since it leads to very low throughput and low network utilization due to an adversarial interaction with delayed ACK support. For fairness, `axtls` on Linux stores files in a ramdisk.

Figure 7(c) depicts the results of running an increasing number of clients making `https` requests. Both Linux and EdgeOS saturate the CPU at around 350 and 700 clients and reach 297K and 668K requests per second, respectively. Similarly, EdgeOS achieves around three times lower 99th percentile latency than `axtls`, and has lower variability across clients compared to Linux as shown by the error bars.

In addition to fast FWP instantiation, and efficient communication, the following FWP optimizations are significant: (1) the FWP abstraction focuses on communication with a *single* client, it avoids event multiplexing through `select` and the associated overhead, and (2) similarly, the share-nothing nature of the FWP abstraction enables synchronization-free

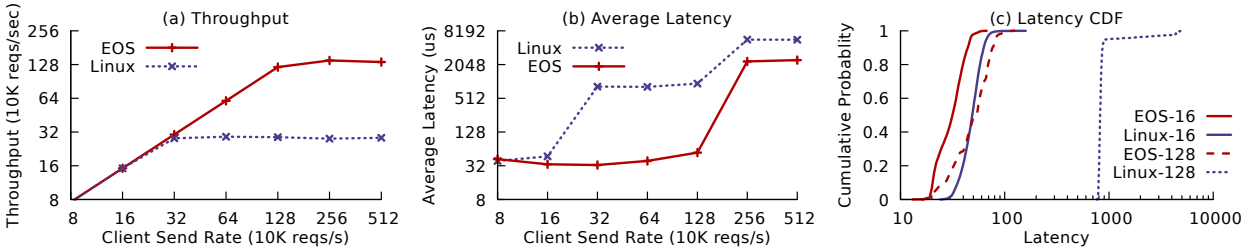

Figure 8: Single memcached instance on one core.

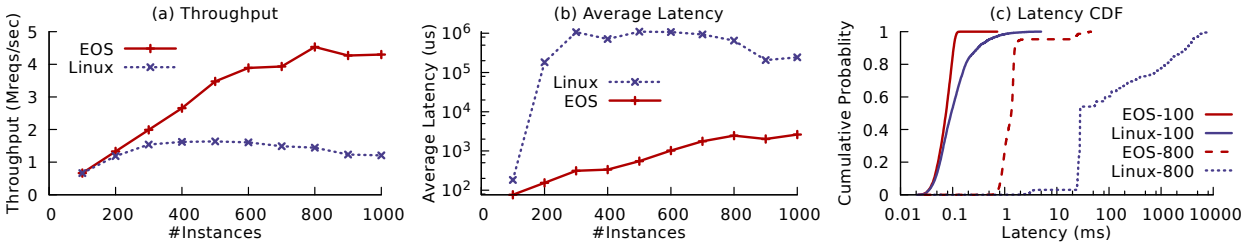

Figure 9: Multiple memcached instances (one per tenant) on 16 cores.

networking using the simple `lwip` stack (in contrast to some unikernels that cannot push `lwip` to Linux-level throughput [32]). Though EdgeOS provides significantly stronger isolation (a TLS instance per client), the FWP model still enables efficient, predictable scalability.

5.5 Edge Inference

Edge based neural network inference enables resource constrained embedded systems to offload computationally-heavy work such as live image recognition. For our second use case we port the CMSIS NN Library (version 1.0.0) neural network inference library to EdgeOS. We use the example CIFAR-10 configuration which takes as input a 32x32 pixel color image which classifies to number 0-9. We focus on providing each tenant with isolated inference services. Thus, we compare an EdgeOS CMSIS NN FWP-per-client against a simple Linux server that `fork`s a process for each client.

	Linux Clients		EdgeOS Clients	
	100	500	100	500
Mean latency (ms)	13.8	69.2	14.6	70.1
99th percentile (ms)	25.4	135	14.8	71.3

We utilize a PowerEdge R740 server with two 24-core Intel Xeon 8160 sockets that represents the Amazon and HPE edge offerings. We use either 100 or 500 clients (separate columns), each requesting 500 inferences. This application is particularly CPU-heavy (each inference taking around 6ms), thus penalizing EdgeOS' design that specializes cores, i.e., Linux can use all cores on the system for inference, whereas EdgeOS sets aside 4 cores for MMA and control services.Despite this, EdgeOS has only slightly lower throughput than Linux, losing 2.5% to 4.6%. However, the simpler FWP runtime in EdgeOS minimizes scheduling interference, reducing latency jitter. Together with EdgeOS's more efficient activation, this yields significant decreases in tail latency – 42% and 47% at 100 and 500 clients, respectively.

5.6 Memcached

Finally, we evaluate how EdgeOS can provide a platform for low latency endpoint applications that don't require rapid instantiation. We implement memcached as an FWP that uses UDP for requests. We mimic a scenario where one or more edge tenants store data, each with many clients making requests. Isolating these tenants from each other is necessary as they should not be able to access (maliciously or not), other tenant's cached data. The EdgeOS controller is used to map incoming requests either to a single memcached FWP (e.g., representing a typical edge cloud data cache) or one FWP per tenant (e.g., representing data stores for different sets of edge-connected IoT devices). We compare EdgeOS against Linux, with a single, or multiple memcached instances. Our workload uses 135 byte value sizes and a 95% get, 5% set request mix generated by the mcblaster client as in [46]. We use multiple 16-core client machines, each running mcblaster processes to ensure the client will not be a bottleneck.

Figure 8 shows a single memcached instance while Figure 9 shows a variant number of instances, representing different edge-cloud tenants. We report the aggregate throughput across all requests, the average latency, and a CDF of the latency (with 16K or 128K clients) to understand the tail. The single instance focuses on the data-path efficiency of the system, while the multi-instance evaluates each system's scalability to an increasing number of tenants on the limited edge hardware.

The efficiency of both systems is seen in their aggregate throughput. EdgeOS processes over 5x the throughput for a single instance, and can scale more gracefully up to 800 instances whereas Linux handles up to 400. EdgeOS' response time for a single memcached instance is substantially lower than Linux: it handles 8X the client request rate before seeing an increase in latency. Since Linux is not able to keep up, it drops a large number of requests, e.g., 5.2% at a 320K req/sec client rate. In contrast, EdgeOS does not see any requests drops at a 1.2M req/sec client rate. For multiple instances,

Linux has a response time of nearly 1 second, whereas EdgeOS has an average latency below 1 millisecond for up to 600 memcached instances. From the latency CDF, we observe that even with only 100 memcached instances, Linux has much higher tail latency than EdgeOS, and that with 800 instances Linux has more than three orders of magnitude worse tail latency. These latency metrics ignore dropped requests – with 800 instances, EdgeOS drops 13% of requests, whereas Linux drops 66%.

6 Related Work

Multi-tenant isolation. Significant research addresses isolation in a multi-tenancy environment. Bolted [38] presented an architecture for a bare metal cloud supporting security sensitive tenants. PSI [70] enables fine-grained and dynamic security postures for different network devices by assigning each device an NF. Denali [68] separates the protection provided by a Virtual Machine Manager (VMM) from the abstractions within a VM, and enables lightweight VM contexts. Multi-tenancy virtual switch designs are proposed in [60, 64]. In contrast, EdgeOS is motivated by the potentially enormous churn and large-scale isolation requirements of the edge cloud, providing service to transient mobile and IoT devices. For isolated edge computation instantiation, FWP compares favorably to forking of minimal Linux processes (two orders of magnitude faster start-time) which is the lower-bound for many such techniques. Re-initializing FWPs to safe states is partially motivated by ChaosMonkey [3, 5].

Lightweight isolation. Wedges [6], LWC [29], and SpaceJMP [16] expand the UNIX interface to include lightweight facilities for controlling and changing protection domains. Similarly, Dune [4] uses hardware virtualization support to provide user-level control over page-tables, and both dIPC [65] and Skybridge [36] use hardware support to bypass the kernel during inter-protection domain communication. Several projects have increased the efficiency of containers. Cntr [63] includes only the application-specific context in a container, while SOCK [48] specializes the container to use efficient kernel operations, and uses a Zygote mechanism paired with a cache to accelerate container creation for stateless computations. EdgeOS targets at abstractions to support immense churn rates, efficient communication with strong isolation via the MMA and a narrow system attack surface. To efficiently use the limited resources in the edge cloud, EdgeOS leverages this support to scale to more than two thousand FWPs in less than 1GB of RAM while maintaining line-rate communication.

Other isolation mechanisms. DMA shadowing [33] utilizes extra memory copies for DMA buffer to provide full IOMMU protection. DAMN [34] introduces DMA-aware packet memory allocator to achieve efficient IOMMU protection. MMA also uses data-copying to avoid shared memory communication between untrusted FWPs. LXDs [42] runs isolated kernel subsystems on dedicated cores. EdgeOS achieves similar isolation with the micro-kernel approach. EdgeOS implements its system-level services in user-level, and further spreads them to different cores.

New hardware features are used to enhance memory isolation, such as Intel MPK [25, 53] and SGX [50, 56]. They are complementary to EdgeOS. Language techniques such as NetBricks [52] implement network processing functions in a memory-safe language. These techniques rely on software isolation within a *single thread*. Without multiplexing the CPU among untrusted FWP chains via preemptive scheduling, *temporal isolation* is challenging. EdgeOS effectively uses the MMA to maintain memory safety, but also provides temporal isolation by executing all FWPs in separate threads that are preemptively scheduled. We also support the direct execution of legacy code modulo the confines of FWP APIs.

7 Conclusions

The increasing prevalence of mobile computations and the Internet of Things requires both scalable isolation facilities for multi-tenancy in the edge, and the agility to handle high churn. This paper has described an optimized copy-based MMA architecture that provides strong mutual isolation without performance penalties. We introduced FWP for scalable isolation that is paired with a cache of post-initialization checkpointed FWP-chains to provide microsecond scale activation times for high churn.

Our evaluation shows EdgeOS substantially improves performance for a wide range of applications from network middleboxes to endpoint services. We show that EdgeOS provides more than a 3.8X reduction in ping latency and more than 2X throughput increase compared to ClickOS – a system that also provides isolated computation – for middlebox computations. More importantly, EdgeOS can create FWPs for client computation in 25-50 microseconds, even when they are created every millisecond, and can scale to over 2000 FWPs while maintaining low latency, even with a very limited amount of memory. For edge applications like memcached, EdgeOS has more than three orders of magnitude decreases in latency when running over 300 server instances simultaneously, and even CPU-intensive TLS termination shows a factor of three tail latency decrease, all while maintaining strong isolation. We believe that EdgeOS paves the way for closely integrating the edge cloud into – and augmenting the capabilities of – the increasing prevalence of mobile and embedded devices.

Acknowledgments. We would like to thank the anonymous reviewers, especially our shepherd Trent Jaeger, for their tremendous feedback that has significantly improved the quality of this paper. We are also thankful to Phani Kishore Gadepalli, Zheng Yang and Runyu Pan for their help on the Composite system. This work was supported by the National Science Foundation under Grants CNS 1815690, CNS 1814234, and CPS 1837382.

References

[1] 5g network slicing in 5gtango, https://www.5gtango.eu/blog/36-5g-network-slicing-in-5gtango.html, 2019.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, 2020.

[3] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), Hollywood, CA, USA, October 8-10*, 2012.

[5] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. *Netflix Tech Blog*, 30, 2012.

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[7] William Earl Boebert and Richard Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[8] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[9] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 26(1):29–35, 1983.

[10] Docker: https://www.docker.com/, 2018.

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

[12] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM. event-place: Vancouver, BC, Canada.

[13] Telecommunications industry association. edge data centers. https://www.tiaonline.org/wp-content/uploads/2018/10/TIA_Position_Paper_Edge_Data_Centers-18Oct18.pdf, 2018.

[14] Micro-data centers out in the wild: How dense is the edge?, https://www.datacenterknowledge.com/archives/2017/05/02/edge-densities, 2017.

[15] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM Press.

[16] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[17] Kevin Elphinstone and Gernot Heiser. 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)*, 2013.

[18] Dawson R. Engler, Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain Resort, Colorado, USA, December 1995. ACM.

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

[20] Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. Temporal capabilities: Access control for time. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017.

[21] The 5g guidea reference for operators the 5g guide: A reference for operators, https://www.gsma.com/wp-content/uploads/2019/04/The-5G-Guide_GSMA_2019_04_29_compressed.pdf, 2019.

[22] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, 2019.

[23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[24] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019.

[26] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 341–354, New York, NY, USA, 2007. ACM.

[27] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Renton, WA, April 2018. USENIX Association.

[28] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[29] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[30] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[31] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11), December 2013.

[32] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[33] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 249–262, New York, NY, USA, 2016. ACM.

[34] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. Damn: Overhead-free iommu protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 301–315, New York, NY, USA, 2018. ACM.

[35] Bishop Matt et al. *Introduction to computer security*, volume 50. Pearson Education India, 2006.

[36] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[37] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, Mountain View CA (USA), 2003.

[38] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting security sensitive tenants in a bare-metal cloud. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[39] Shahid Mumtaz, António Morgado, Kazi Huq, and Jonathan Rodriguez. A survey of 5g technologies: Regulatory, standardization and industrial perspectives. *Digital Communications and Networks*, 2017.

[40] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

[41] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical difc enforcement on android. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[42] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. Lxds: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019.

[43] NGMN Alliance, 5G End-to-End Architecture Framework, 2017.

[44] NGMN Alliance, 5G White Paper, 2017.

[45] NGMN Alliance, Description of Network Slicing Concept, 2017.

[46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[47] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 1–14, New York, NY, USA, 2017. ACM.

[48] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[49] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira. Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges. *IEEE Communications Magazine*, 55(5):80–87, 2017.

[50] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

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

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

[53] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019.

[54] Gabriel Parmer and Richard West. Predictable interrupt management and scheduling in the Composite component-based system. In *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS'08), Barcelona, Spain, November 30 - December 3*, 2008.

[55] Larry Peterson. Cord: Central office re-architected as a datacenter. *Open Networking Lab white paper*, 2015.

[56] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[57] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[58] P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, and H. Bakker. Network slicing to enable scalability and flexibility in 5g mobile networks. *IEEE Communications Magazine*, 2017.

[59] Sandvine. The Global Internet Phenomena Report, October 2018.

[60] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[61] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP'99), Kiawah Island Resort, South Carolina, USA, December 12-15*, 1999.

[62] Yuqiong Sun, Giuseppe Petracca, Xinyang Ge, and Trent Jaeger. Pileus: Protecting user resources from vulnerable cloud services. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (CCS)*, 2016.

[63] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[64] Kashyap Thimmaraju, Saad Hermak, Gabor Retvari, and Stefan Schmid. MTS: Bringing multi-tenancy to virtual networking. 2019.

[65] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems (Eurosys)*, 2017.

[66] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53. ACM, December 1995.

[67] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. Speck: A kernel for scalable predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15), Seattle, WA, USA, April 13-16*, 2015.

[68] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications, 2002.

[69] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[70] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. PSI: precise security instrumentation for enterprise networks. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[71] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, Berkeley, CA, USA, 2008.

[72] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 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*. ACM, August 2016.