Understanding Network Startup for Secure Containers in Multi-Tenant Clouds: Performance, Bottleneck and Optimization

Paper #295, 16 pages

ABSTRACT

In this paper, we use empirical measurements to show that container network startup is a key factor that contributes to the slow startup of secure containers in multi-tenant clouds, especially in the scenario of serverless computing where the issue is pronounced by high-volume concurrent container invocations. We conduct extensive and detailed analysis on existing Container Network Interface (CNI) solutions and show that even the best one doubles the startup time from the no-network scenario. We show that the major cause of the blowup in total startup time is that enabling network significantly increases the contention among different startup stages, particularly for global locks including the Routing Table NetLink (RTNL) mutex lock and various spin locks. We reveal that contending for these locks hinders startup performance in three ways, including directly increasing stage time, causing poor pipeline overlap and wasting CPU resources. We propose and evaluate two preliminary solutions, virtual device pooling and Bayes-based concurrency control, to optimize the startup time. Our results show that we are able to reduce the container startup time by 58.9%, facilitating fast and high-volume secure container invocations.

1 INTRODUCTION

Secure containers, such as Firecracker from AWS [2], Kata Containers from OpenStack [40] and RunD from Alibaba Cloud [49], provide strong isolation for applications in multi-tenant clouds, and have attracted significant attention in recent years. A notable use case for these secure containers is in serverless services, like function compute (e.g. AWS Lambda [4]) and NoSQL database (e.g. Azure Cosmos [10]). These services typically feature short-lived application instances, with lifecycles spanning just seconds. Cloud vendors will face severe resource waste if the invocation time takes up a large portion. Furthermore, these applications often come with stringent Service Level Objectives (SLOs) that demand tight control over invocation time. Meanwhile, given the potential for serverless applications to experience surges in demand, e.g. over 200 container invocation requests arriving near simultaneously on an Alibaba serverless platform node [49], it not only calls for fast but also high-volume secure container invocations.

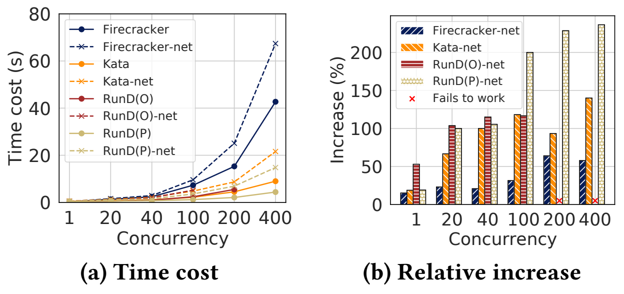


Figure 1: Concurrent startup time of different secure containers with and without network. RunD(O) and RunD(P) are the open-source and production versions of RunD respectively. Startup time of RunD(O) with concurrency of 200 and 400 are missing as RunD(O) fails to work in such cases. Takeaway: (a) Enabling network significantly increases startup time. (b) The relative increase becomes larger with the no-network startup time being further optimized (from Firecracker to RunD(P)).

Recent work have focused on optimizing secure containers to achieve fast concurrent invocations, but without incorporating network capabilities [2, 49]. Container-based applications often necessitate network communication to interact with other cloud services, such as AWS's S3 storage service [5]. In multi-tenant clouds, in addition to the isolation provided by secure containers for resources like CPU and memory, network isolation is also imperative. Mainstream cloud providers today like AWS[5], Azure[58] and GCP[32], all provide such network isolation using virtual Switch (vSwitch). This arrangement of secure containers in conjunction with vSwitch is recognized as the modern container infrastructure. We find that despite existing optimizations, none of the systems conforming to the modern container infrastructure can truly achieve both fast and high-volume invocations, and the container network startup is a key factor that causes the trouble. Fig. 1 shows our preliminary measurements about the overhead of the multi-tenant container network under different concurrency settings. The container network is built with a classic ipvlan-based CNI plugin [22] and a widely-used VXLAN-enabled vSwitch named *Open vSwitch* (OVS) [66]. The ipvlan-based CNI has the shortest startup time among CNIs that are compatible with all the different

	Net. Startup Performance	Secure Container	vSwitch-based Multi-tenancy
Bankston *et al.* [13], KAPOČIUS *et al.* [38], Novianti *et al.* [61] Lim *et al.* [50], Zhao *et al.* [85], Mentz *et al.* [57],Atici *et al.* [7] Suo *et al.* [70], Zeng *et al.* [84], Park *et al.* [65], Boeira *et al.* [16]	×	×	×
Suo *et al.* [72]	×	×	✓
Kumar *et al.* [44], Wang *et al.* [77]	×	✓	×
Suo *et al.* [71], Qi *et al.* [68]	✓	×	×
Our work	✓	✓	✓

Table 1: Existing measurement work on container networks. None of the work evaluates the concurrent startup performance in modern container infrastructure.

secure containers. We observe that, when starting up with a high concurrency of 400 containers, all secure containers experience a significant increase in their end-to-end startup time. As RunD(P) has a better optimized startup time tailored for the no-network scenario, its relative time increase incurred by enabling network is more pronounced, which goes as high as 263%.

The need to optimize container network concurrent startup thus necessitates a comprehensive understanding of the performance of the various existing CNI solutions and the answering of the following questions: *What is the container concurrent startup time in modern container infrastructure using different types of CNI? What are the bottlenecks and their causes? From what aspects can we relieve the bottlenecks and reduce the startup time?*

Unfortunately, we find that existing measurement work on container networks cannot well answer these questions. The majority of existing work do not consider the modern container infrastructure. A secure container contains an independent Operating System (OS) and is in fact a type of micro Virtual Machine (VM). It requires extra virtualization to pass the network devices to the container. Along with the setup of vSwitch, enabling network becomes more complex and costly than that in the traditional container case considered by existing work, and its startup performance and bottlenecks have not been thoroughly studied. Tab. 1 summarizes the existing measurement work, most of them only measure the data plane performance of different CNIs, trying to find a way to achieve similar latency, bandwidth and throughput as the host network [7, 13, 16, 38, 50, 57, 61, 65, 70, 84, 85]. Several of them consider secure containers [44, 77] or vSwitch-enabled multi-tenancy[72] in their data plane measurements, but neglect the discussion of startup performance. Suo *et al.* [71] and Qi *et al.* [68] measure the startup time of different CNIs, but neither consider modern container infrastructure nor dive deep into the startup process. Therefore, in this paper, we evaluate the detailed startup process of container networks in modern container infrastructure and conduct extensive measurements to identify the performance, bottlenecks and potential optimization opportunities. Our major contributions are summarized as follows:

(1) We provide a comprehensive evaluation of main CNI types with secure container in terms of concurrent startup time, data plane performance and resource consumption. We find that ipvtap-based CNI has the best overall performance. It ties for first place in data plane performance, and achieves 11.5% shorter startup time than the second best, with less than 10% increase in host CPU consumption. However, even with ipvtap, enabling network still doubles the startup time from the no-network scenario.
(2) We perform an in-depth analysis of the startup process and identify the global locks, particularly the `RTNL` lock and the `spin` locks as the common bottleneck of all different CNIs. Enabling network significantly increases the contention among different startup stages, especially for these global locks. We reveal that such contention increases stage time, causes poor pipeline overlap and wastes CPU resources, and thus blows up the total startup time.
(3) We propose and evaluate two preliminary solutions to optimize the startup process, including eliminating the contention with CNI pre-creation and mitigating the effect of contention by using concurrency control to improve pipeline overlap and reduce CPU waste. We design and implement a virtual device pool and a Bayes-based concurrency control mechanism. Our evaluation demonstrates that a full pre-creation of CNI reduces the overall startup time by 49.8% at a high memory cost, while the concurrency control mechanism reduces the startup time by 18.8% at almost no cost. By combining the two techniques, we achieve a total startup time reduction of 58.9%.
(4) We will open source the entire software implementations of our measurement tools and optimization tools as well as our test dataset.

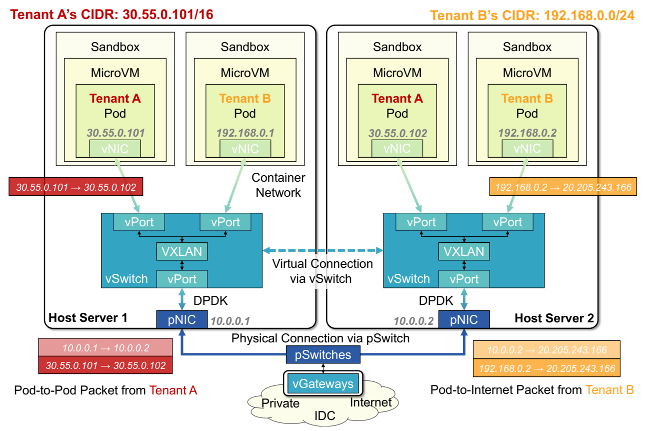


Figure 2: vSwitch enabled container network.

2 BACKGROUND

2.1 Container network in multi-tenant clouds

Nowadays major cloud providers such as AWS, Azure and GCP all offer multi-tenancy capability based on vSwitch, which isolates packets from different tenants with tunnel protocols such as VXLAN [55] and Geneve [80]. Fig. 2 exemplifies how vSwitch provides multi-tenant container networking, where tenant A and tenant B (marked with red and orange color) each build containers within their own overlay virtual networks, and the vSwitch can be regarded as a layer-2 bridge with several virtual ports (vPort) bridging the virtual NICs (vNIC) of containers and the physical NICs (pNIC) of host servers.

For pod-to-pod networking, tenant A encodes a packet with the source address `30.55.0.101` of a pod in host 1 and the destination address `30.55.0.102` of another pod in host 2. The vSwitch in host 1 first ingresses the original packet through the connection constructed by container network interface (CNI) plugins [22] between the pod's vNIC and the vSwitch's vPort. Then, it encapsulates the packet with an overlay header with the source address `10.0.0.1` of host 1 and destination address `10.0.0.2` of host 2 and egresses it to the vSwitch in host 2 through the physical network infrastructure. The rest of the data path goes the inverse way of decapsulation and decoding. For pod-to-Internet networking, tenant B encodes a packet destined for a public service over the Internet. The vSwitch on host 2 recognizes the public address `20.205.243.166` of that packet and encapsulates it with host 2's IP address `10.0.0.2` for sending through the virtual gateway (vGateway).

Note that vSwitch is not the only way to enable a multi-tenant container cloud. Taking community CNI plugins for example, Flannel utilizes the Linux Kernel VXLAN as tunneling-based isolation [29], Calico leverages user-defined routes to separate packets from multiple tenants [18], and Cilium relies on high-performance eBPF hackings to boost the performance of the Linux Kernel VXLAN. However, cloud providers insist on the vSwitch-based solution for the following reasons. First, vSwitch offers the VXLAN-enabled network isolation, which is more secure than user-defined routes based isolation. Second, it provides the most abundant network control features like Quality-of-Service (QoS) guarantee and security group rules. As vSwitch, such as OVS, has strong open-source community support, adopting vSwitch can better benefit from the features and performance optimizations brought by the open-source community. Third, other cloud services like databases and computation use vSwitch-based network, adopting the same solution allows tenants' containerized applications to connect to these services seamlessly. In addition, for data plane performance, CNIs built on vSwitch VXLAN can obtain reduced times of user-kernel space traversing than those built on Linux Kernel VXLAN. In line with the choice of current mainstream cloud providers, this paper mainly discusses the network cold start procedure for vSwitch-enabled container networks, especially those built on OVS.

2.2 Network cold start for secure container

Traditional containers built solely on process-level isolation technologies like `cgroup` and `namespace` suffer from security problems of privilege escalation, information disclosure and covert channels [2]. Therefore, cloud providers choose secure containers when launching their cloud-native services. Two main technical routes for secure containers are investigated. The first category like Google gVisor [35, 82] restricts shared kernel functions and thus avoids unsafe operations. The others like Kata Containers [40], AWS Firecracker [2] and AliCloud RunD [49] run containers inside lightweight micro VMs for isolation. The latter is more widely adopted in both the industry and the open-source community for a better balance between performance and security [6], and it is the focus of our work.

As shown in Fig. 2, the state-of-the-art secure containers are built with two layers of the isolation stack [49]. The first layer is the *sandbox* environment, it leverages `cgroup` and `namespace` to allocate hardware resources among secure containers. The second layer, denoted by *microVM*, virtualizes a lightweight guest OS to run the tenant's containers. To enable network, the CNI plugin must provide a complete data path that passes through the full stack from host, sandbox, and microVMs to container. We dig into the source code of Kata Containers and summarize the three-step end-to-end procedure of network cold start in secure containers in Fig. 3. Note that the underlying logic of the procedure is consistent across other secure containers like Firecracker and RunD.

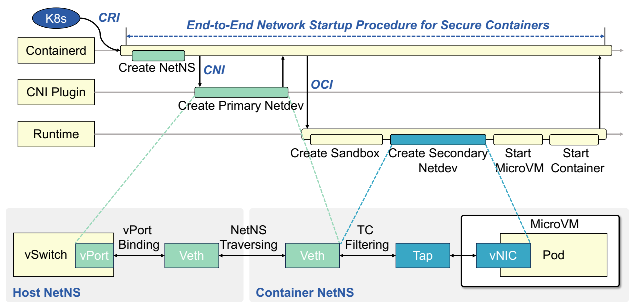


Figure 3: Detailed procedure of network cold start in secure container framework.

(1) NetNS Creation: After receiving a creation command from the container orchestrator like Kubernetes (K8s) [42] through the standardized interface like container runtime interface (CRI), the container engine like Containerd [24] first creates a new container network namespace (NetNS). This NetNS is isolated from the host NetNS and other container NetNSs in terms of network devices (NetDev) and configurations like routing tables and access control policies.

(2) Primary NetDev Creation: Then, the container engine invokes the CNI plugin to create the primary NetDevs. The purposes of the primary NetDevs are to traverse the host and container NetNS, and provide a NetDev in the container NetNS, referred to as the CNI endpoint, to wait to be connected to the container. Fig. 3 shows the primary NetDevs created by the vSwitch-enabled CNI plugin with *Veth Direct* mode, in the green color. A pair of `veth` NetDevs transfer packets between two NetNSs. The one in the container NetNS is the CNI endpoint, and the CNI plugin allocates a valid IP for it. The one in the host is bound to a vPort in vSwitch, and the vSwitch is then configured with network policies to satisfy user requirements such as Quality of Service (QoS) and security group rules.

(3) Secondary NetDev Creation: Last, the container engine invokes the container runtime through the standardized interface like open container initiative (OCI) to create the secondary NetDevs which is used to traverse the host kernel and the guest kernel. In Kata Containers, a `tap` NetDev is created in the container NetNS which connects the `veth` through Linux traffic control (TC) filter. TC filter is a feature in the Linux operating system that enables fine-grained traffic and traffic manipulation. It can connect two NetDevs by simply redirecting traffic between them. Finally, the QEMU hypervisor virtualizes and passes the `tap` to MicroVM.

Only when both the primary and secondary NetDevs are correctly created and connected can the pod visit its container network. Compared with the traditional container infrastructures considered in existing work where network setup only includes the creation of NetNSs and the pair of `veth` NetDevs, the existence of secure container and vSwitch further complicates the setup process, and the startup performance and bottlenecks in such case remain to be illustrated.

Primary NetDev	CNI plugins
`veth`	Flannel[29], Antrea[1], ovs-cni[63], Kube-OVN[41], Weave Net[78], Kube-router[43], AWS bridge CNI[8], GCP Kubenet CNI[31], Azure Overlay CNI[11], Azure Kubenet CNI[11]
`eBPF-based veth`	Cilium[20], Calico[18], Isovalent[36], Azure eBPF CNI[11], GCP Dataplane V2[31]
`macvlan & ipvlan`	Tungsten Fabric[74], DANM[25]
`vhost-user`	Kube-OVN[41], AWS VPC CNI[8], GCP VPC CNI[31], Azure VNET CNI[11], Intel Userspace CNI[76]

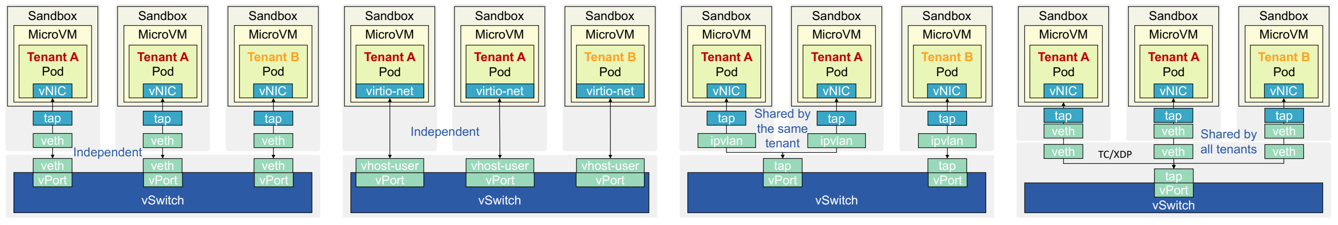
Table 2: Summarization of multi-tenant CNI plugins.

3 MEASUREMENT METHODOLOGY

3.1 Selected CNIs for measurement

To select representative CNI plugins, we have surveyed solutions [1, 18, 20, 29, 36, 41, 43, 74, 78] from the cloud-native community, *i.e.,* CNCF project landscape for the container network [21], cloud providers including AWS, Azure and GCP [8, 11, 31], and several corresponding open-source projects [63, 76]. The key differences of the various CNIs boil down to how they implement the primary NetDevs and multi-tenancy. We classify the existing CNIs in Tab. 2 based on the type of the primary NetDevs, of which there are mainly four types, i.e. `veth`, `ipvlan` & `macvlan`, `vhost-user` and `eBPF`-based `veth`. For multi-tenancy implementation, as we have discussed in §2.1, the mainstream providers adopt OVS, and their implementations of the various CNIs are, in fact, the use of the corresponding primary NetDevs on top of OVS, like the Azure version of Cilium [9]. Therefore, our work follows the same implementation approach and combines the four types of primary NetDevs with OVS to represent the various CNIs shown in Tab. 2.

Recall that binding primary NetDevs to vPorts and configuring the vSwitch is an important network setup stage introduced by the modern container infrastructure, and it can be a potential startup performance bottleneck. Generally, the time cost of this stage is related to the tenant sharing granularity of vPorts and the number of network rules configured. Our work intends to measure and discuss the effect of these factors. As shown in Fig. 4, the tenant sharing granularity of vPorts can be summarized as the following modes:



(a) **veth direct.** (b) **vhost direct.** (c) **Single-tenant sharing.** (d) **Multi-tenant sharing.**

Figure 4: Different implementations of multi-tenancy for vSwitch-based CNIs.

(1) Direct mode: In this mode, one vPort of a vSwitch serves only one pod of a certain tenant without any sharing mechanism. Fig. 4a and Fig. 4b depict two types of the direct mode implemented with a pair of `veth` NetDevs and a pair of `vhost-user` and `virtio-net` NetDevs [67] respectively. Their main differences are that: (i) `veth` is based on full host kernel network stack, while `vhost-user` uses memory sharing techniques to bypass several host kernel operations. (ii) `veth` requires the help of secondary NetDevs to traverse the guest kernel, while the `virtio-net` NetDev paired with `vhost-user` can be directly passed to the guest.

(2) Single-tenant sharing mode: In this mode, one vPort connects all pods of a single tenant, as shown in Fig. 4c. Compared to the direct mode, vPort sharing can efficiently reduce the number of vPorts, thus it reduces the resource usage and potentially also the startup time. But it has to share the vPort-specific network policy, *e.g.,* QoS, among all pods of the tenant. `ipvlan` is the standard Linux NetDev that implements data-path sharing of a NetDev, and another choice for single-tenant sharing is `ipvtap` NetDev, in which a `tap` is hooked on one `ipvlan`'s MAC layer to transfer packets. Since the `tap` of the `ipvtap` can be directly passed to the guest, the secure container runtime doesn't need to create an additional `tap` and its TC-based connection hook with the primary NetDev for the packet traversing. We also include `ipvtap` as a type of CNI in our measurement.

(3) Multi-tenant sharing mode: In this mode, all pods of all tenants share just one vPort, as shown in Fig. 4d. Such global sharing among all tenants requires offloading packet routing and network policy execution from the vSwitch to the CNI plugins. Usually, the XDP hook controlled by the injected eBPF program and the TC hook controlled by the `iproute2` user interface can satisfy these requirements [20]. This mode is expected to have the least overhead of the vPort creation. Thus besides the major CNI types in Tab. 2, we also include a multi-tenant sharing CNI called `tc-routing` as illustrated by Fig. 4d.

3.2 Measurement testbed setup

Hardware setup. Our testbed includes two servers, each server has a 2-socket NUMA-enabled Intel Xeon Platinum 8369B 2.9GHz CPU with 32 physical cores per socket, 80KB/1280KB/48MB L1/L2/L3 caches, 512GB RAM, and a 200Gbps Mellanox ConnectX-6 pNIC. We conduct the concurrent startup experiment on a single server as the invocation of containers on different servers does not affect each other. We also measure the data-plane performance, i.e. pod-to-pod network communication intra- and inter-servers, alongside resource consumption to ensure that CNIs with faster startup speeds do not have significant deficiencies in data plane performance and resource consumption. For inter-server network communication, we connect two servers' pNICs directly through an optical fiber.

Software setup. The servers run CentOS 8 operating system with Linux kernel v4.18.0. The container and network software include Containerd v1.7.3 [24], Kata Containers v3.2.0 [40] and OVS v3.1.2 [30]. Recall that in Fig. 1, Kata secure container has worse startup performance compared with RunD, thus it does not seem to be the best choice. However, the industry version of RunD, i.e. RunD(P), is closed-source and the open-source version, i.e. RunD(O), fails to work when the startup concurrency reaches 200. Therefore, our performance measurement and bottleneck analysis are based on Kata, which is the best-performing one with both open-source code and the support for high startup concurrency. Our findings on Kata can be generalized to other secure containers as their network setup processes follow the same logic. We implement the vSwitch-enabled multi-tenant CNI plugins summarized in §3.1 by integrating open-source CNI projects [20, 23, 63, 76] with OVS and secure containers. We use the *Kata-qemu* configuration and assign a memory of 512MB for each container. The containers use the default microVM kernel (v5.19.2) and image (alpine 3.15) provided by Kata. In our measurements, the startup of a container is considered successful when the microVM finishes launching with its network enabled and accessible from other containers. We maintain a persistent container that performs pinging to check the network accessibility of the microVMs.

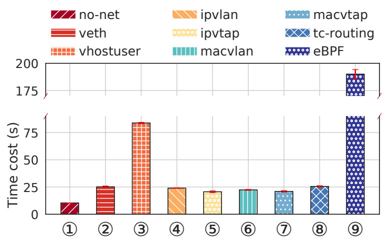


Figure 5: Container concurrent startup time with different CNI types (concurrency = 400).

3.3 Performance metrics and tools.

Concurrent startup performance. Assuming a concurrency of N secure containers, we define the metric of concurrent startup time as the elapsed time from when the concurrent startup command is issued to when the last of the N containers successfully starts. A shell script is responsible for simultaneously starting N processes to initiate container startup requests to the Containerd daemon, and counting the end-to-end time consumption. We use $N = 400$, aligning with the configuration in the RunD paper [49], which is based on statistical analysis from the Alibaba serverless platform. To break down the startup procedure and identify bottlenecks, we develop a time logging tool and integrate it with the container engine, runtime and CNI plugins to inspect the detailed time consumption of each component. This tool employs memory caching and asynchronous file writing techniques to keep the additional time overhead introduced by the logging process to less than 3%, thereby ensuring minimal impact on the overall time.

As mentioned in §3.1, the setup and configuration of the vSwitch introduces variables like the number of user-defined QoS, security group rules, and the number of tenants that can impact the total startup time. For the main measurements in §4.1.1, we assume that only one tenant exists and no QoS and security group rules are applied. Then we study the impact of these variables in §4.1.2.

Data-plane performance and resource consumption. We allocate 4 vCPU cores to each container, use `iperf3` [37] to collect throughput (Gbps), and use the RR (Request-Response) test in `netperf` [60] to measure network latency (μs). We report CPU utilization of both the host and the secure container during startup and communication, respectively. The CPU utilization is collected using the `psutil` library of `python3.6`.

4 CONCURRENT NETWORK STARTUP PERFORMANCE AND BOTTLENECKS

4.1 Measured startup performance

4.1.1 Concurrent startup time.

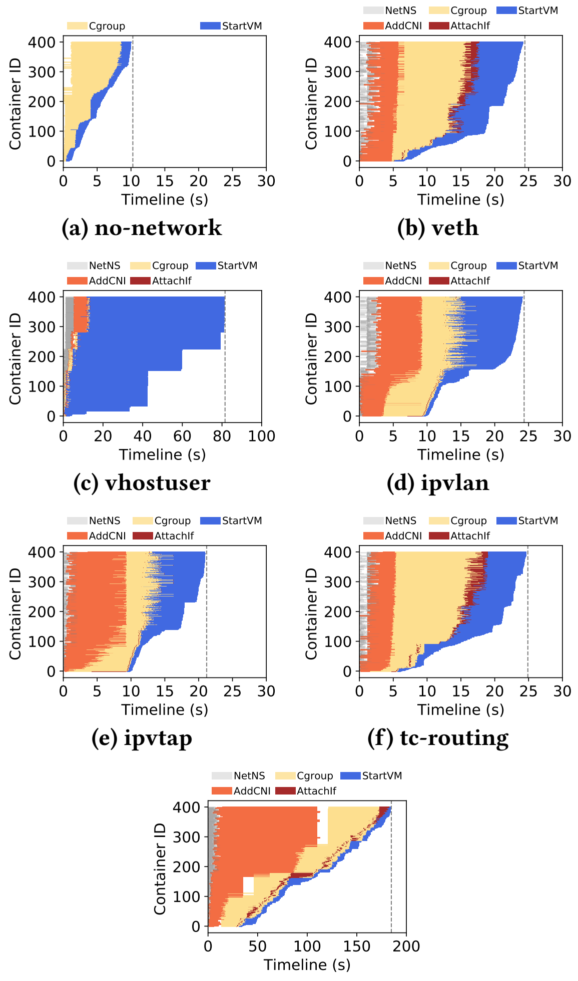
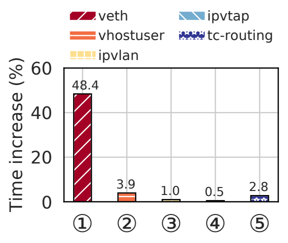


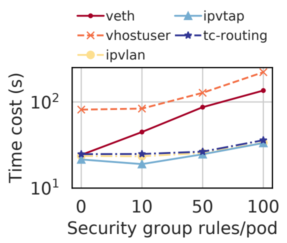
Figure 6: Startup time breakdown.

Overall performance. Fig. 5 shows the concurrent startup time of each CNI type. We can observe that all CNI types incur obvious startup time overhead compared with no-network startup (①). `eBPF` achieves the worst performance, increasing the startup time by 18X. `Ipvtap` and `macvtap` achieve the best performance, however, they still double the total startup time compared with no-network startup, indicating the need to optimize the network startup process. Note that the performance of `macvtap` is nearly identical to that of `ipvtap` because they share similar setup procedures and data paths. The same goes for `macvlan` and `ipvlan`. Thus we will ignore `macvlan` and `macvtap`, and focus on `ipvlan` and `ipvtap` in the rest of this paper.

Time consumption breakdown. Fig. 6 shows the time breakdowns of startup processes in different conditions. Of the five stages shown in Fig. 6, `NetNS` refers to creating new

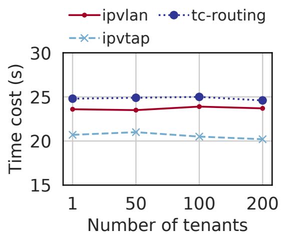


(a) QoS.

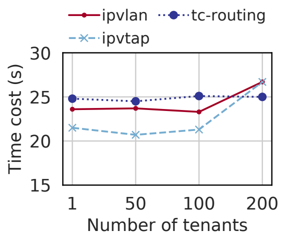


(b) Security group rules.

Figure 7: Impact of enabling QoS and security group rules on container startup time.



(a) No QoS rule.



(b) With QoS rules.

Figure 8: Impact of the number of tenants on container startup time.

network namespaces, `AddCNI` means creating primary NetDevs and configuring vSwitch, `Cgroup` refers to isolating container resources with Linux cgroup, `AttachIf` refers to creating and setup secondary NetDevs, and `StartVM` means creating microVM. These five stages constitute more than 90% of the total startup time. We find that, first, the overhead of enabling network does not only come from the network setup operations as illustrated in Fig. 3, but also largely from the increased time of other stage not directly related to network setup, i.e. `Cgroup`. We will show in §4.2 that such bottleneck comes from the contention for global locks. Similarly, contention for locks is the underlying reason for the noticeable time increase in `StartVM` stage of `vhostuser`.

Second, by comparing (b) and (e), we see that the superior performance of `ipvtap` mainly comes from the reduction of `Cgroup` time. One might expect `ipvtap` to achieve a shorter `AddCNI` time since `ipvtap`, as the sharing mode CNIs, reduces the number of created vports on OVS. However, its `AddCNI` time is much larger compared with that of `veth`. That is because although `ipvtap` reduces the vport creation time, the creation of ipvtap device itself incurs a larger overhead than creating veth pairs and thus increases the overall `AddCNI` time.

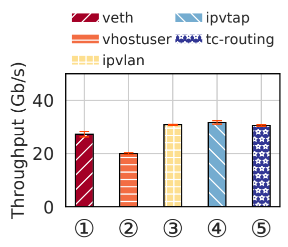
Third, poor startup performance of `eBPF` is caused by the enlarged `AddCNI` time. We find that an eBPF program requires recompilation at each startup process to implement its tenant subordination and corresponding net policy, causing high time costs. Such results suggest that eBPF-based CNIs are currently unsuitable for supporting the ever-growing short-lived serverless applications, so we neglect their exploration in the following sections.

Takeaway 1: *Existing CNIs all incur significant time overhead to the concurrent startup of secure containers. The overhead comes from both the time costs of introduced network operations and the time increase of other stages like operating cgroup.*

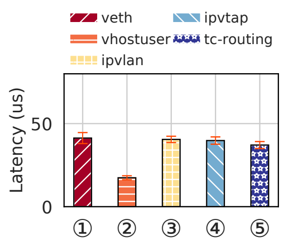
4.1.2 Impact of variables on startup time.

QoS and security group rules. We configure two QoS rules for each vport to control the average and peak bandwidth of the ingress and egress traffic. Results in Fig. 7a show that enabling user QoS incurs a 48.4% startup time increase on `veth` but small increase (< 4%) on other CNIs. For `ipvlan`, `ipvtap` and `tc-routing`, all containers share a vport on OVS, thus the QoS rules are only configured for one port and incur small overhead. `vhostuser` uses the `dpdkvhostuser` vport type in OVS, and its QoS rules are implemented using `DPDK` library instead of Linux TC like other types of vports. The `DPDK` library has high configuration efficiency and incurs small overhead. Unlike QoS rules that are configured at vport level, security group rules are configured for each pod using OVS flow tables. We increase the number of rules per pod from 0 to 100. Results in Fig. 7b show that adding security group rules increases the startup time of `ipvlan`, `ipvtap` and `tc-routing` only when the number of rules per pod exceeds 10. This is because the insertion of security group rules does not cause contention with other startup stages and can be hidden in the startup pipeline. It only affects the overall performance when its time cost becomes dominant in the pipeline. We will further analyze the pipeline in §4.2.1.

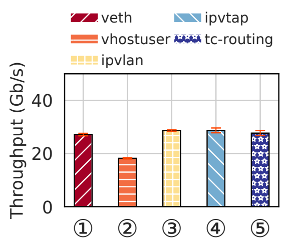
Number of tenants. Fig. 8a demonstrates the effect of the number of tenants on the startup time of the sharing mode CNIs. Essentially, the number of tenants served on one server determines the number of vports created on OVS for single-tenant sharing CNIs, i.e. `ipvlan` and `ipvtap`, but it does not affect the number of vports for multi-tenant-sharing CNI, i.e. `tc-routing`. We increase the number of tenants from 1 to 200, and distribute the 400 containers evenly to each tenant. It should be noted that the mappings between tenants and containers have no impact on startup time, a fact that we have confirmed. This is because such mappings influence neither the number of vports nor the number of NetDevs. The results show that, when no QoS rule is enabled, the startup time of `ipvlan` and `ipvtap` does not increase with the number of tenants. This is because the time cost of creating more vports



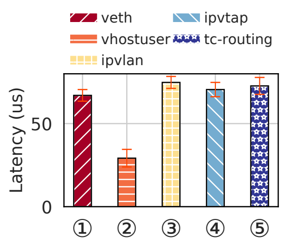
(a) Intra-node throughput



(b) Intra-node latency

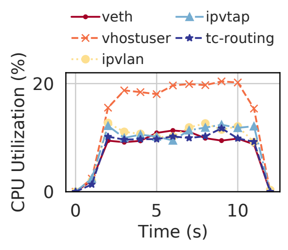


(c) Inter-node throughput

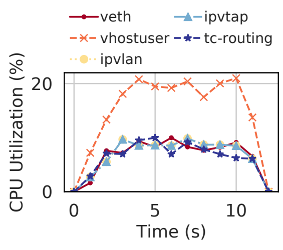


(d) Inter-node latency

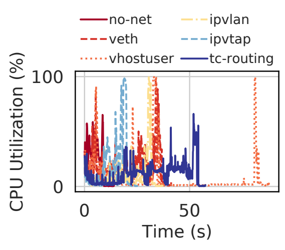
Figure 9: Data-plane performance of different CNIs.



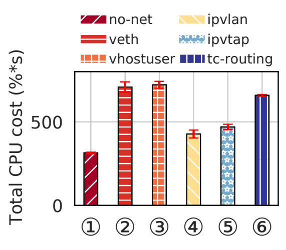
(a) CPU utilization during intra-node communication.



(b) CPU utilization during inter-node communication.



(c) Host CPU utilization timeline during startup.



(d) The sum of host CPU utilization during startup.

Figure 10: Resource consumption of different CNIs.

is small and can be hidden by other startup stages. When QoS rules are enabled, increasing the number of tenants causes more QoS rules to be configured for `ipvlan` and `ipvtap`. Since configuring QoS rules has a relatively larger cost than creating vports, it incurs noticeable overhead on the startup time. As a result, the startup time of `ipvlan` and `ipvtap` slightly exceeds that of `tc-routing` as the number of tenants reaches 200.

Takeaway 2: *Tenant sharing granularity affects the number of vports and QoS rules on vSwitch. The time cost of creating vports is minor and not a determining factor of startup time. In contrast, configuring QoS rules can incur noticeable overhead on single-tenant sharing CNIs when the number of tenants is large, in which case the multi-tenant-sharing CNI* `tc-routing` *can slightly outperform the former.*

4.1.3 *Data-plane performance and resource consumption.*

Fig. 9 shows the intra-node and inter-node data-plane performance of different CNIs. In general, CNIs of the sharing mode (③-⑤) achieve similar throughput and latency. For intra-node communication (Fig. 9a), the direct mode CNIs (①-②) have relatively lower throughput because their intra-node traffic needs to traverse vSwitch, while the traffic of sharing mode CNIs can be directly forwarded by shared vports. Performing inter-node communication slightly degrades the throughput compared with intra-node communication (degradation < 10%), but significantly increases the communication latency. It is noticeable that `vhostuser` has lower latency than others, as the primary NetDev of `vhostuser` can be virtualized and directly passed to the MicroVM without having to rely on secondary Netdevs (demonstrated in Fig. 4b), which shortens the communication path.

We report containers' CPU utilization during intra and inter-node communications in Fig. 10a and Fig. 10b. The CPU utilization of `vhostuser` is relatively higher while those of other CNIs are at a similar level. Fig. 10c shows the change in host CPU utilization during the concurrent startup process. We also sum the utilization in each second and report the results in Fig. 10d to show the total CPU consumption. We observe that `ipvlan` consumes the least CPU resources and the consumption of `ipvtap` is only slightly higher.

Takeaway 3: *With data plane performance and resource consumption considered, we can conclude that ipvtap-based CNI achieves the best startup time for secure containers with less than 10% increase in host CPU consumption compared with the second best CNI type.*

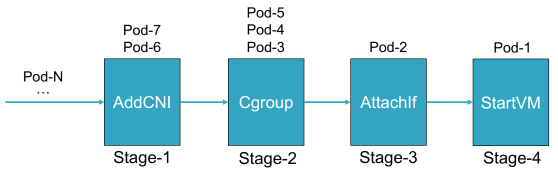


Figure 11: Pipeline model of the concurrent startup process. Containers (Pods) pass through each stage to perform the startup operation and the overall time is dominated by the most time-consuming stage.

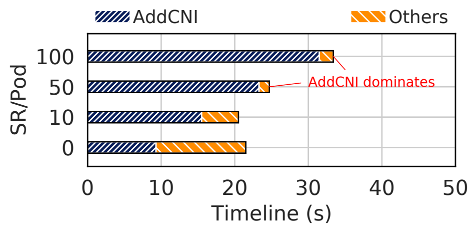


Figure 12: Breakdown of startup time with different number of Security group Rules per Pod (SR/Pod).

4.2 Bottleneck analysis

As existing CNIs incur high overhead to the startup process, it is important to understand bottlenecks and seek further optimizations. In this section, we analyze the bottlenecks from a pipeline perspective. We divide the effect of enabling network as non-competitive and competitive time overhead, and show how the contention for locks in the pipeline increases the competitive time overhead in three ways.

4.2.1 Non-competitive time overhead. The concurrent startup process can be regarded as a pipeline model. Fig. 11 shows four stages in the startup pipeline as an example. The containers pass through each stage to perform the corresponding startup operation, and the total time cost is determined by the stage with the highest time cost. As an example, we break down the startup time of `ipvtap` from Fig. 7b and report the results in Fig. 12. The results display the time when `AddCNI` stage finishes inserting security group rules to vSwitch and when the whole startup process ends. Inserting more Security group Rules (SR) per pod increases the `AddCNI` time, but the total time is barely affected at first, *i.e.,* SR/Pod=0,10. This is attributed to that `AddCNI` itself is not the dominant stage, and the insertion of security group rules operates `OVS` flow tables which does not cause contention with other stages, thus the increased time can be overlapped behind other stages. Similar examples include the numerous device configuration operations during `StartVM`, which, having no contention with other stages like `AddCNI`, `cgroup`, etc., can also be overlapped in pipeline. We refer to time overhead like this as non-competitive time overhead. Only when the non-competitive time overhead becomes dominant of the pipeline, *i.e.,* SR/Pod=50,100, it incurs an obvious increase in the overall startup time.

	Times of obtaining RTNL lock	RTNL lock contention time
no-network	803	162 ms
veth	19992	9.8 s
ipvlan	16542	11.7 s
ipvtap	6105	9.2 s
tc-routing	18859	14.6 s

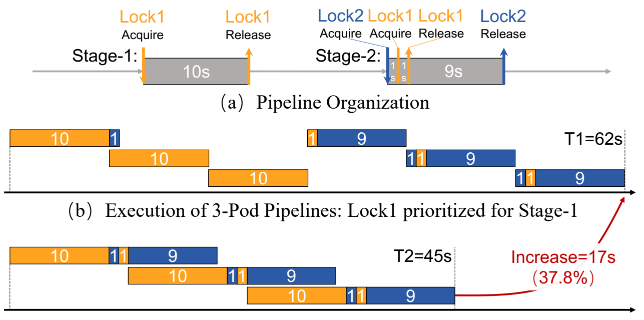
Table 3: `RTNL` lock obtaining times and total contention time. Concurrent startup without network also obtains `RTNL` lock as it also setups the `netprio` cgroup.

Takeaway 4: *The concurrent startup process is a pipeline model. The time overhead of network setup operations can be overlapped behind other pipeline stages if those operations neither are the dominant stage nor contend with other stages.*

4.2.2 Competitive time overhead. When network setup operations contend with other stages and increase startup time, we refer to the additional time as competitive time overhead. We dig into the cause of the contention and identify that global locks, particularly the `RTNL` mutex lock and various `spin` locks, are the ones to blame.

The `RTNL` mutex lock in Linux systems is a specialized synchronization mechanism designed to protect the Linux networking data structures during operations that alter the networking configuration, such as creating or removing network devices. Its utilization is crucial in maintaining the integrity and performance of network operations within the Linux kernel. The operations to enable network for secure container severely increase the contention for the `RTNL` lock. As illustrated in Tab. 3, when concurrently launching 400 containers, even with the best-performing CNI, `ipvtap`, the `RTNL` lock is obtained 6105 times in total, increased by 7.6X compared with no-network startup. Its `RTNL` lock contention time covers 9.2s of the total startup time, and over 59.0% (62.3% for `veth`, 74.7% for `ipvlan` and 69.7% for `tc-routing`) of the `RTNL` lock obtaining operations are issued by the `AttachIf` and `StartVM` stages where the primary NetDev and secondary NetDev are connected, and the secondary NetDev is virtualized to the microVM. Those operations are introduced by secure containers and make enabling network more costly compared with that for traditional containers.

The first way `RTNL` lock contention affects the competitive time overhead is the direct time increase of related stages.



(a) Pipeline Organization

(b) Execution of 3-Pod Pipelines: Lock1 prioritized for Stage-1

(c) Execution of 3-Pod Pipelines: Lock1 prioritized for Stage-2

Figure 13: Example of two stages contending for global locks. Time cost is labeled on the block. Prioritize the obtaining of Lock1 in stage-2 reduces the overall time.

A typical example is the `Cgroup` shown in Fig. 6. Compared with the no-network case in Fig. 6a, the use of `ipvlan` CNI in Fig. 6d greatly increases the `Cgroup` time and contributes the most to the time overhead. The cause is that the setup of `netprio` type of cgroup requires the `RTNL` lock and thus suffers from contention. The contention also accounts for why `ipvtap` achieves better startup performance, since the endpoint created by `ipvtap` is a `tap` NetDev which can be directly passed to microVM, it requires neither additional secondary NetDevs nor TC filter to connect. As creating secondary NetDevs and adding TC filter rules also require the `RTNL` lock, avoiding these operations reduces the contention. In contrast, `tc-routing` not only uses TC filter rules to connect the endpoint and the secondary NetDevs, but also sets up additional rules to route the traffic from the vport to each container, further increasing the contention and leading to its poor startup performance.

> ***Takeaway 5:*** *Enabling network for secure containers incurs severe contention for the global* `RTNL` *lock and significantly increases the time of stages involving the lock, like operating cgroup.*

The second way `RTNL` lock contention increases competitive time overhead is that the uncontrolled order of obtaining lock leads to poor pipeline overlap. Fig. 13 is a simple example to illustrate how the order of obtaining the same lock in different pipeline stages affects the overall time. Assuming 3 containers are traversing a two-stage pipeline, as shown in Fig. 13(a), the first stage requires obtaining Lock1, e.g. the global `RTNL` lock, and the second stage requires first obtaining Lock2, e.g. the global `cgroup_mutex` lock, and then obtaining Lock1. This example corresponds to how locks are obtained when setting up `netprio` group. The best order is to prioritize obtaining Lock1 in stage 2 as shown in Fig. 13(c), which leads to a total time cost of 45s. However, when two stages contend for Lock1 freely, the obtaining order is uncontrolled. In the worst-case scenario shown in Fig. 13(b) where the obtaining of Lock1 in stage 1 is prioritized, there is nearly no overlap between the two pipeline stages. As a result, the total time becomes 62s, increased by 37.8%.

> ***Takeaway 6:*** *The uncontrolled order of obtaining global locks by different stages leads to poor pipeline overlap and increases the overall startup time.*

The third way the competitive time overhead is increased is that the contention for `spin` locks wastes CPU resources and thus stalls the startup. Unlike other locks that put the waiting thread to sleep, a `spin` lock causes the thread to continuously check (or "spin") for the lock to become available. This is efficient if threads are only expected to hold the lock for a short period of time, but can be wasteful of CPU resources if held for longer durations. When a large amount of containers starts concurrently, they contend for the global spin locks in network configuration and cgroup operations like the `css_set_lock`, and can spend a lot of time spinning on the locks. By using eBPF to sample the lock spinning time, we discover that enabling network increases the spinning time by at least 105% with the different CNIs, resulting in more CPU waste. Our subsequent experiment in §5.2 will demonstrate that mitigating spin lock contention through concurrency control can effectively reduce CPU consumption and achieve overall startup time reduction.

> ***Takeaway 7:*** *The contention for* `spin` *locks causes increased CPU waste and contributes to the competitive time overhead.*

Based on the major bottlenecks analyzed above and how they affect the startup time, we discuss the possible optimization approaches in the next section.

5 ACCELERATING CONCURRENT NETWORK STARTUP FOR SECURE CONTAINERS

The optimization solutions can be categorized based on whether they are implementation-specific or more general techniques. Implementation-specific optimization requires analyzing the specific operations in each stage and improving their efficiency. For the major bottleneck, i.e. the contention of global locks, the implementation-specific optimization is to redesign the locks with the corresponding data structures they protect. However, such redesign is typically difficult to perform [79] for system-wide locks like the *RTNL* lock.

Our focus is more on general optimization techniques. Centering around the three ways that global lock contention

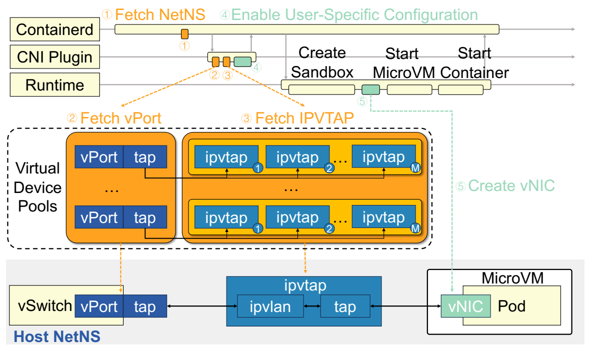


Figure 14: Virtual device pooling. The user-free virtual devices are fetched from precreated pools (orange colored) and user-specific ones are created in real time (green colored).

leads to competitive overhead, *i.e., direct time increase of the contended stage*, *poor pipeline overlap* and *CPU waste* as mentioned in §4.2.2, we first evaluate the effectiveness of *virtual device pooling* technology in eliminating the contention and discuss its memory cost (§5.1). When the memory resource is scarce, we propose and evaluate another technology named *multi-stage concurrency control* to mitigate contention by balancing the startup pipeline and reducing the CPU waste (§5.2). The concurrency control technique boosts the concurrent startup performance in a very lightweight way and can further work with pooling to reduce startup time.

5.1 Virtual device pooling

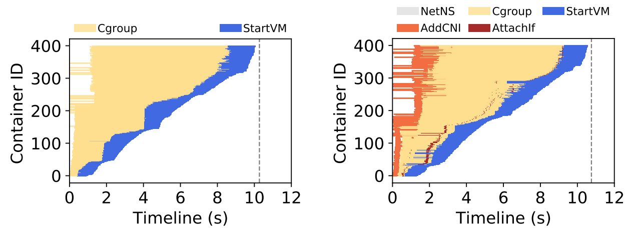
Pooling is a common technology that accelerates time-consuming operations at the expense of certain precreated resources [14, 28, 52, 59, 75]. We explore pooling to eliminate the stage contention from the source, and our contributions are to (i) explore which stages in the network startup of secure containers can be pooled, (ii) verify the upper limit of the performance gain brought by this technology, and (iii) point out its shortcomings to inspire subsequent works.

Design: Taking the `ipvtap` CNI as an example, Fig. 14 illustrates whether each detailed stage can be pooled in the network startup procedure. First, the NetNS is obtained from a pool (①). Then, the two main NetDevs, `vPort` and `ipvtap` are fetched instantly from their pools (② and ③) when the CNI plugin is invoked. One `vPort` belonging to one tenant's VXLAN group is connected with M `ipvtap`, where M stands for the maximum number of secure containers that can be created for one tenant. Finally, the operations involving user-specific configurations such as IPAM and network policy appliance (④) and newly created container instances such as vNIC creation (⑤) are left to run in real time.

Implementation: We implement a pooling agent with a simple HTTP interface. The pooling agent invokes OVS and NetLink interfaces for the precreation. One pooling agent is deployed on each physical server to serve the local concurrent fetching requests, which is scalable to the future clusterized deployment. In the implementation of the NetNS pool, since the container NetNS inside the microVM is already isolated, we can avoid creating the excessive NetNS outside the microVM (as shown in Fig. 3) and maintains a global NetNS, *i.e.,* the host NetNS in the pool. It should be noted that the speedup in time comes from the pooling operation rather than the reduced number of NetNS.

Results: Fig. 15b shows that the time cost of concurrent network startup with virtual device pooling is only 10.5s. Compared to the non-pooling result of 20.7s in Fig. 6e, it achieves a significant performance gain of 49% time reduction. Moreover, the time cost is comparable to that when the network is disabled (shown in Fig. 15a) with an overhead within 5%. The corresponding time breakdowns illustrate that, although there still are operations that cannot be pooled in advance, other pipeline stages can hide their time costs.

Discussions: Pooling achieves significant performance gain but can incur large memory overheads. As the number of tenants scheduled to the server and the number of containers of each tenant are not known in advance, it's challenging to predetermine the total number of vports or the requisite quantity of ipvtaps for each vport. The straightforward solution is to create 400 vports and each with 400 `ipvtaps`. We observe that a memory cost as high as 50GB is incurred, which is a high cost to the cloud provider. Therefore, we call for subsequent research work to improve the efficiency of pooling. Potential solutions like demand prediction [15, 33, 59] for each user or identifying the best trade-off between pooled and non-pooled stages can be further explored.



(a) No-network. **(b) `Ipvtap` pre-creation.**

Figure 15: Startup time breakdown. Precreation effectively eliminates the time overhead. The time difference is within 5% compared with no-network startup.

5.2 Multi-stage concurrency control

In some scenarios where memory resource for running container network is strictly controlled, *e.g.,* bare metal servers, it is difficult to deploy optimization technologies like pooling. Therefore, we expect to explore techniques that can achieve

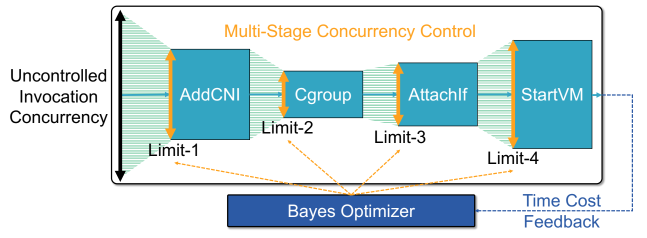
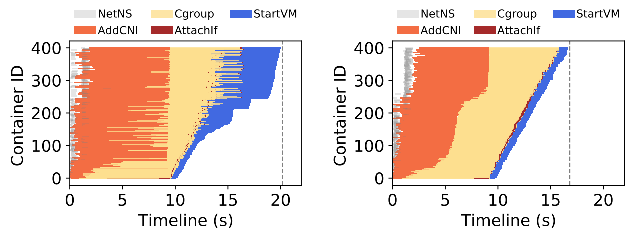


Figure 16: Design of multi-stage concurrency control.



(a) w/o concurrency control. **(b) w/ concurrency control.**

Figure 17: Bayes-based concurrency optimization results. Total startup time is reduced by 18.8%.

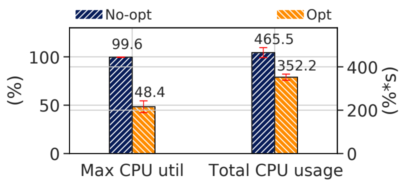


Figure 18: CPU consumption with concurrency control. Concurrency control reduces both the maximum CPU utilization and the total CPU usage.

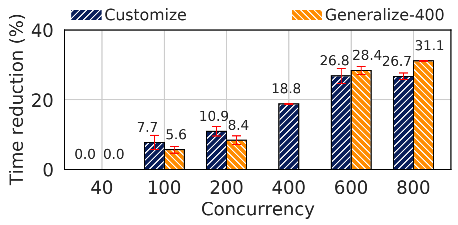


Figure 19: Startup time reduction with concurrency control. Customize: run Bayes directly with each concurrency. Generalize-400: applying parameters obtained with 400 concurrency to other concurrencies.

a noticeable performance gain but with lower resource costs. The observation of poor pipeline overlap (Takeaway 6) and CPU waste (Takeaway 7) problems inspires us to apply proactive concurrency control to schedule how the different stages obtain locks and reduce unnecessary wait on the `spin` locks.

Design: As shown in Fig. 16, we divide the startup process into 4 major stages according to our previous timeline breakdown and contention analysis. We apply different concurrency limits on each of the stages. The general idea is to control the number of containers in each stage to adjust the priority of obtaining locks and avoid unnecessary wait on the `spin` locks. To identify these concurrency limits, we adopt a classic black-box optimization technology called *Bayes optimization* [17], which is commonly used in hyperparameter determination in machine learning. The Bayes optimizer iteratively adjusts the concurrency limits based on the posterior probability of the end-to-end feedback and optimizes the overall startup time.

Implementation: The concurrency control is implemented with a locally centralized agent similar to that in the previous subsection. The agent uses semaphores based on HTTP requests and replies to control the number of containers entering each stage. The Bayes optimizer is trained offline with a fixed concurrency of 400 containers, and we adopt the best set of parameters after 50 iterations. The optimization averagely takes 3 hours which is an acceptable cost.

Results: Fig. 17a and Fig. 17b depict the startup time breakdown of `ipvtap` when disabling and enabling concurrency control. The end-to-end startup time is reduced by 18.8%, from 20.9s to 17.0s. We observe a significant reduction in the CPU consumption of the startup process as shown in Fig. 18, indicating that concurrency control effectively reduces CPU waste.

Discussions: The most common concern about learning-based optimization methods is generalizability. We believe that changes in hardware setup or the CNI implementations do require re-training. But the learned parameters are supposed to generalize to different concurrency settings, as demonstrated in Fig. 19. When applying the concurrency control parameters trained with 400 containers to test cases of other concurrencies, the time reductions are still obvious. And the gaps to the results of running Bayes optimization directly with each concurrency are relatively small. Note that the results of running Bayes directly with 600 and 800 concurrencies are slightly worse than those of applying the parameters of 400 concurrencies. This is because higher concurrency enlarges the search space of Bayes optimization and makes it more difficult to learn. Another issue worth discussing is the splitting granularity of the concurrency-controlled stages. Fine granularity can potentially improve the optimization performance but also expands the search space of Bayes optimization and increases the difficulty of learning. We leave exploring more fine-grained concurrency control algorithms as a future research topic.

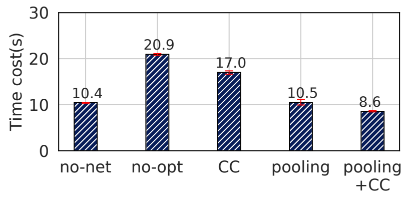


Figure 20: Startup time (400 containers) using different techniques. Combining pooling with concurrency control (represented by CC) further accelerates the startup.

5.3 Pooling + Concurrency Control

Apart from being an alternative that optimizes start time at a low cost, our concurrency control technique can also be combined with pooling to further accelerate the startup process. The optimization results using different techniques are summarized in Fig. 20. Although pooling already reduces the startup time to a comparable level as that without network, concurrency control still achieves a time reduction of 18.1% when applied. Such improvement results from that the no-network startup also suffers similar problems like CPU waste caused by the contention for `spin` locks. Overall, our combined techniques reduce the concurrent startup time of network-enabled containers by 58.9%.

6 RELATED WORKS

In §1, we have introduced the related works about the measurement part in detail and illustrated our work's position. Thus, we mainly discuss the related works about the optimization part in the following.

Data-plane optimization of container networks. In recent years, plenty of works have shown interest in pushing the data-plane performance of container networks to the limit bounded by the host network [19, 46, 47, 51, 87]. Several works try to optimize CNI data plane's pipeline parallelism [47] or its resource allocation [46], while the others manage to avoid the VXLAN's overhead by designing a connection-level overlay with either kernel optimization [87] or eBPF hacking [19, 51]. However, none of these works consider network startup performance, and some even sacrifice it as shown by our previous analysis for `eBPF` based CNIs.

Speeding up the startup of containers or VMs. The majority of related works focus on the startup performance of containers but without any network connection [27, 48, 53, 62, 69, 81, 86]. They either reduce cold startup performance by accelerating container image distribution [48, 53] and introducing specific checkpoint or general template-based runtime [27, 62], or provide the warm startup solutions with technologies like workload prediction and adaptive pooling [69, 81, 86]. LightVM [56] and Bacou et al. [12] optimize the boot time of VMs, but they neglect either the effect of enabling network or the effect of concurrent startup. Few works optimize the network startup of containers[59, 75]. PCPM pre-creates multiple pause containers with the network connection and manages the assignment and recycle dynamically [59]. Particle [75] analyzes the network startup process for traditional containers and utilizes network namespace sharing and IP sharing technique to reduce the time cost of moving NetDevs across network namespaces. However, it is not the major bottleneck with secure containers.

Lock optimization. Locks are crucial to application performance and their optimization has gained the attention of many works. The most effective way to optimize lock contention is to redesign the corresponding data structures the locks protect and reduce or even remove the use of locks. Such an approach is typically application specific, and thus a series of work [3, 54, 73, 83] only focus on providing tools to profile the behavior or identify the bottlenecks of locks, and leave the redesign to application developers. However, Westphal et al. [79] digs into the `RTNL` lock and shows that it is difficult to redesign or remove. Another series of works propose more general approaches to optimize the performance of kernel locks [26, 34, 39, 64]. Kashyap et al. [39] and Dice et al. [26] reduce the cost of accessing locks by making the locks NUMA-aware, TClocks [34] reduces the cost by avoiding the transfer of lock-guarded shared data. Those works are orthogonal to our concurrency control method. SynCord [64] abstracts key behaviors of kernel locks and performs advanced scheduling policies on them, but it requires the co-design of user applications. Hybrid Lock [45] reduces resource usage by dynamically switching between `mutex` and `spin` locks with eBPF, however, it also incurs the modification in both user-space programs and kernel functions.

7 CONCLUSION

In this paper, we use extensive empirical measurements to show that network startup is one of the key factors that contribute to the overall time of secure container startup. We dive into this problem and show that enabling network significantly increases the contention among different startup stages, especially for globals locks including the `RTNL` mutex lock and `spin` locks, and thus becomes the bottleneck of achieving fast and high-volume secure container invocation. We propose and explore several solutions, including virtual device pooling and Bayes-based concurrency control, to address this problem. We believe that the issues we discovered are insightful for future research in the area of secure container-based applications.

A ETHICS

This work does not raise any ethical issues.

REFERENCES

[1] ACNI. 2023. Antrea CNI. https://github.com/antrea-io/antrea.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of USENIX NSDI*. 419–434.

[3] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of ACM EuroSys*. 298–313.

[4] Amazon. 2023. AWS Lambda. https://www.aliyun.com/product/fc.

[5] Amazon. 2023. Cloud Computing Services - Amazon Web Services. https://aws.amazon.com/.

[6] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending Containers and Virtual Machines: A Study of Firecracker and gVisor. In *Proceedings of ACM SIGPLAN/SIGOPS VEE*. 101–113.

[7] Gulsum Atici and Pinar Sarisaray Boluk. 2020. A Performance Analysis of Container Cluster Networking Alternatives. In *Proceedings of International Conference on Industrial Control Network And System Engineering Research*. 10–17.

[8] AwsCNI. 2023. AWS CNI Collection. https://docs.aws.amazon.com/eks/latest/userguide/managing-vpc-cni.html.

[9] Azure. 2023. Azure CNI with Cilium. https://azure.microsoft.com/en-us/blog/azure-cni-with-cilium-most-scalable-and-performant-container-networking-in-the-cloud/.

[10] Azure. 2023. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db.

[11] AzureCNI. 2023. Azure CNI Collection. https://learn.microsoft.com/en-us/azure/aks/configure-azure-cni?tabs=configure-networking-portal.

[12] Mathieu Bacou, Grégoire Todeschi, Daniel Hagimont, and Alain Tchana. 2019. Nested virtualization without the nest. In *Proceedings of IEEE International Conference on Parallel Processing*. 1–10.

[13] Ryan Bankston and Jinhua Guo. 2018. Performance of Container Network Technologies in Cloud Environments. In *Proceedings of IEEE International Conference on Electro/Information Technology*. 0277–0283.

[14] Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, and Jordy Dieltjens. 2021. Reducing cold starts during elastic scaling of containers in kubernetes. In *Proceedings of ACM Symposium on Applied Computing*. 60–68.

[15] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of ACM SoCC*. 153–167.

[16] Conrado Boeira, Miguel Neves, Tiago Ferreto, and Israat Haque. 2021. Characterizing Network Performance of Single-Node Large-Scale Container Deployments. In *Proceedings of IEEE International Conference on Cloud Networking*. 97–103.

[17] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[18] Calico. 2023. Calico CNI Project. https://github.com/projectcalico/calico.

[19] Sunyanan Choochotkaew, Tatsuhiro Chiba, Scott Trent, and Marcelo Amaral. 2022. Bypass Container Overlay Networks with Transparent BPF-driven Socket Replacement. In *Proceedings of IEEE International Conference on Cloud Computing*. 134–143.

[20] Cilium. 2023. Cilium CNI Project. https://github.com/cilium/cilium.

[21] CNCF. 2023. CNCF Cloud Native Network. https://landscape.cncf.io/card-mode?category=cloud-native-network&grouping=category.

[22] CNCF-Foundation. 2023. CNI (Container Network Interface), a Cloud Native Computing Foundation Project. https://www.cni.dev/.

[23] CNI. 2023. CNI Plugins project. https://github.com/containernetworking/plugins.

[24] Containerd. 2023. Containerd: An Industry-Standard Container Runtime with An Emphasis on Simplicity, Robustness and Portability. https://containerd.io/.

[25] DCNI. 2023. DANM CNI. https://github.com/nokia/danm.

[26] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In *Proceedings of ACM EuroSys*.

[27] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of ACM ASPLOS*. 467–481.

[28] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. 2022. Dgsf: Disaggregated gpus for serverless functions. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*. 739–750.

[29] Flannel. 2023. Flannel CNI Project. https://github.com/flannel-io/flannel.

[30] Linux Foundation. 2023. Open vSwitch Project. https://www.openvswitch.org/.

[31] GcpCNI. 2023. GCP CNI Collection. https://cloud.google.com/kubernetes-engine/docs/concepts/dataplane-v2.

[32] Google. 2023. Google Cloud Computing Services. https://cloud.google.com/.

[33] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of ACM Middleware Conference*. 280–295.

[34] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. 2023. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLOCKS. In *Proceedings of USENIX OSDI*. 1–16.

[35] GVisor. 2023. gVisor Documentation. https://gvisor.dev/docs/.

[36] ICNI. 2023. Isovalent CNI. https://isovalent.com/.

[37] iPerf. 2023. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr.

[38] Narūnas Kapočius. 2020. Performance Studies of Kubernetes Network Solutions. In *Proceedings of IEEE Open Conference of Electrical, Electronic and Information Sciences*. 1–6.

[39] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and practical locking with shuffling. In *Proceedings of ACM SOSP*. 586–599.

[40] Kata. 2023. Kata Containers: the Speed of Containers, the Security of VMs. https://katacontainers.io/.

[41] KCNI. 2023. Kube-OVN CNI. https://github.com/kubeovn/kube-ovn.

[42] Kubernetes. 2023. Kubernetes: An Open-Source System for Automating Deployment, Scaling and Management of Containerized Applications. https://kubernetes.io/.

[43] KuCNI. 2023. Kube-router CNI. https://github.com/cloudnativelabs/kube-router.

[44] Rakesh Kumar and B. Thangaraju. 2020. Performance Analysis between RunC and Kata Container Runtime. In *Proceedings of IEEE International Conference on Electronics, Computing and Communication Technologies*. 1–4.

[45] Victor Laforet, Jean-Pierre Lozi, and Julia Lawall. 2023. BPF Hybrid Lock: Adaptive Synchronization for Multi-Core Processors. In *Proceedings of ACM SOSP Poster)*. 1–8.

[46] Kyungwoon Lee, Kwanhoon Lee, Hyunchan Park, Jaehyun Hwang, and Chuck Yoo. 2022. Autothrottle: satisfying network performance requirements for containers. *IEEE Transactions on Cloud Computing* (2022), 1–13.

[47] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. 2021. Parallelizing packet processing in container overlay networks. In *Proceedings of ACM EuroSys*. 1–16.

[48] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-Level Image Service for Agile and Elastic Application Deployment. In *Proceedings of USENIX ATC*. 727–740.

[49] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-Concurrency Startup in Serverless Computing. In *Proceedings of USENIX ATC*. 53–68.

[50] Sang Boem Lim, Joon Woo, and Guohua Li. 2020. Performance Analysis of Container-Based Networking Solutions for High-Performance Computing Cloud. *International Journal of Electrical and Computer Engineering* 10, 2 (2020), 1507–1514.

[51] Shengkai Lin, Peirui Cao, Tianyi Huang, Shizhen Zhao, Quan Tian, Qi Wu, Donghai Han, Xinbing Wang, and Chenghu Zhou. 2023. XMasq: Low-Overhead Container Overlay Network Based on eBPF. *arXiv preprint arXiv:2305.05455* (2023).

[52] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. 2021. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proceedings of ACM SOSP Workshop on Programming Languages and Operating Systems*. 38–45.

[53] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. 2019. CFS: A Distributed File System for Large Scale Container Platforms. In *Proceedings of ACM SIGMOD*. 1729–1742.

[54] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. 2019. LockDoc: Trace-based analysis of locking in the Linux kernel. In *Proceedings of ACM EuroSys*. 1–15.

[55] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. 2023. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://www.rfc-editor.org/rfc/rfc7348.html.

[56] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of ACM SOSP*. 218–233.

[57] Lucas Litter Mentz, Wilton Jaciel Loch, and Guilherme Piêgas Koslovski. 2020. Comparative Experimental Analysis of Docker Container Networking Drivers. In *Proceedings of IEEE International Conference on Cloud Networking*. 1–7.

[58] Microsoft. 2023. Microsoft Azure: Cloud Computing Services. https://azure.microsoft.com/.

[59] Mohan, Anup and Sane, Harshad and Doshi, Kshitij and Edupuganti, Saikrishna and Nayak, Naren and Sukhomlinov, Vadim. 2019. Agile Cold Starts for Scalable Serverless. In *Proceedings of USENIX HotCloud*. 1–6.

[60] Netperf. 2023. Netperf. https://hewlettpackard.github.io/netperf/.

[61] Siska Novianti and Achmad Basuki. 2021. The Performance Analysis of Container Networking Interface Plugins in Kubernetes. In *Proceedings of International Conference on Sustainable Information Engineering and Technology*. 231–234.

[62] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of USENIX ATC*. 57–70.

[63] OVS. 2023. Open vSwitch CNI plugin project. https://github.com/k8snetworkplumbingwg/ovs-cni.

[64] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2022. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*. 667–682.

[65] Youngki Park, Hyunsik Yang, and Younghan Kim. 2018. Performance Analysis of CNI (Container Networking Interface) Based Container Network. In *Proceedings of IEEE International Conference on Information and Communication Technology Convergence*. 248–250.

[66] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The Design and Implementation of Open vSwitch. In *Proceedings of USENIX NSDI*. 117–130.

[67] QEMU. 2023. Vhost-user Protocol. https://qemu-project.gitlab.io/qemu/interop/vhost-user.html.

[68] Shixiong Qi, Sameer G. Kulkarni, and K.K. Ramakrishnan. 2020. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability. *IEEE Transactions on Network and Service Management* 18, 1 (2020), 656–671.

[69] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming Serverless Functions Better with Heterogeneity. In *Proceedings of ACM ASPLOS*. 753–767.

[70] Kun Suo, Yong Shi, Ahyoung Lee, and Sabur Baidya. 2021. Characterizing Networking Performance and Interrupt Overhead of Container Overlay Networks. In *Proceedings of ACM Southeast Conference*. 93–99.

[71] Kun Suo, Junggab Son, Dazhao Cheng, Wei Chen, and Sabur Baidya. 2021. Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing. In *Proceedings of IEEE International Conference on Cluster Computing*. 433–443.

[72] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An Analysis and Empirical Study of Container Networks. In *Proceedings of IEEE INFOCOM*. 189–197.

[73] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. 2010. Analyzing lock contention in multithreaded applications. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 269–280.

[74] TCNI. 2023. Tungsten Fabric CNI. https://github.com/tungstenfabric/tf-controller.

[75] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: Ephemeral Endpoints for Serverless Networking. In *Proceedings of ACM SoCC*. 16–29.

[76] UCNI. 2023. Userspace CNI. https://github.com/intel/userspace-cni-network-plugin.

[77] Xingyu Wang, Junzhao Du, and Hui Liu. 2022. Performance and Isolation Analysis of RunC, gVisor and Kata Containers Runtimes. *Springer Cluster Computing* 25, 2 (2022), 1497–1513.

[78] WCNI. 2023. Weave Net CNI. https://github.com/weaveworks/weave.

[79] Florian Westphal. 2017. RTNL mutex, the network stack big kernel lock. *Proceedings of Netdev* 2 (2017).

[80] Wikipedia. 2023. Generic Network Virtualization Encapsulation. https://en.wikipedia.org/wiki/Generic_Network_Virtualization_Encapsulation.

[81] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proceedings of ACM ASPLOS*. 768–781.

[82] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *Proceedings of USENIX HotCloud*. 1–6.

[83] Tingting Yu and Michael Pradel. 2016. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of ACM*

SIGSOFT International Symposium on Software Testing and Analysis. 389–400.

[84] Hao Zeng, Baosheng Wang, Wenping Deng, and Weiqi Zhang. 2017. Measurement and Evaluation for Docker Container Networking. In *Proceedings of IEEE International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*. 105–108.

[85] Yang Zhao, Nai Xia, Chen Tian, Bo Li, Yizhou Tang, Yi Wang, Gong Zhang, Rui Li, and Alex X. Liu. 2017. Performance of Container Networking Technologies. In *Proceedings of ACM SIGCOMM HotConNet Workshop*. 1–6.

[86] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. Aquatope: QoS-and-Uncertainty-Aware Resource Management for Multi-Stage Serverless Workflows. In *Proceedings of ACM ASPLOS*. 1–14.

[87] 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 *Proceedings of USENIX NSDI*. 331–344.