Vol.2022-OS-156 No.9
2022/7/27

Accelerating TCP/IP Communications in Rootless Containers by Socket Switching

Naoki Matsumoto[1,a)] Akihiro Suda[2,b)]

Abstract: ”Rootless containers” is a concept to run the entire container runtime and containers without the root privileges. It protects the host environment from attackers exploiting container runtime vulnerabilities. However, when rootless containers communicate with external endpoints, the network performance is very low compared to rootful containers because of the overhead of the user-land TCP/IP implementation called ”slirp4netns”.
In this paper, we propose ”bypass4netns” that accelerate TCP/IP communications in rootless containers by bypassing slirp4netns. bypass4netns uses sockets allocated on the host. It switches socket file descriptors in containers to the host's socket file descriptors by intercepting syscalls and injecting the file descriptors, using ioctl(`SECCOMP_IOCTL_NOTIF_ADDFD`). We confirmed that rootless containers with bypass4netns can achieve more than 10 times faster throughput than rootless containers without it.

Keywords: Rootless Containers, Linux Network Namespaces, Seccomp User-space Notification

1. Introduction

In modern computing resource sharing, it is common to separate the execution environment in terms of security and convenience. There are some methods to isolate the execution environment. A common one is to use virtual machines(VMs). VMs virtualize the entire machine. Therefore, this method has overhead from virtualization itself and from running an OS on each VM. The other method is containers. Containers achieve the isolation by separating userland processes with Linux kernel features. They have the advantage of higher speed performance and more efficient use of computing resources compared to VMs.

A software stack that provides containers using Linux kernel features is called a ”Container Runtime”. There are some container runtimes such as Docker[1] and Podman[2]. They provide not the only container itself but also utilities to create and distribute container images.

Processes in a container cannot manipulate the host because the processes in the container run in an environment isolated from the host. The isolation is provided by Linux kernel features like namespaces[3] and capabilities[4]. Therefore, a process in a container cannot perform operations including malicious operations on the host side without explicit configurations. However, vulnerabilities in the container runtime or Linux kernel may allow processes in a container to manipulate the host side[5]. In this case, since the container runtime is running with root privileges, the vulnerability in the container runtime can be used to perform malicious operations or run a malicious code with root privileges on the host side. CVE-2019-14271[6], which allows a malicious NSS library to be loaded, is known as such vulnerability.

Rootless containers[7] is a container runtime improving the security of containers by running itself without root privileges. Even if the runtime has vulnerabilities, malicious operations or malicious codes are not executed with root privileges. Rootless containers use some methods like user namespaces to enable a container runtime to run without root privileges. However, there are some problems caused by these methods. One of them is low network performance. When container runtime runs without root privileges, the networking system of container runtime with root privileges cannot be used as it is. Therefore, networking components called RootlessKit[8] and slirp4netns[9] are used to provide the same functionality in an unprivileged environment. The networking performance for external endpoints with these components is low.

In this paper, we propose bypass4netns, a method to bypass the communication performance bottleneck in rootless containers and provide network functionality with equivalent performance to that of a rootful container. bypass4netns uses socket switching to bypass the communication bottleneck from inside a container to the outside and from the outside to the inside of the container. Socket switching is to switch a socket created in the process in the container to a socket created on the host. Socket switching is accomplished by dynamically handling system calls with Seccomp User-space Notification(Seccomp Notify)[10]. If socket switching is needed, sockets are switched with Seccomp Notify's SECCOMP_IOCTL_NOTIF_ADDFD. We implemented bypass4netns and confirmed that the throughput is 10 times faster than that of rootless containers without bypass4netns.

The following is the structure of this paper. Section 2 describes the background of rootless containers, Section 3 describes the de-

[1] Kyoto University
[2] NTT Software Innovation Center
[a)] mt2.naoki@net.ist.i.kyoto-u.ac.jp
[b)] akihiro.suda.cz@hco.ntt.co.jp

sign of bypass4netns, Section 4 describes the analysis of socket behavior, Section 5 describes the implementation of the prototype system and the evaluation, Section 6 shows the performance evaluation results, Section 7 discusses the security Consideration and the limitations of bypass4netns, and Finally, Section 8 summarizes this paper.

2. Backgrounds

2.1 Rootless Containers

In ordinary containers, their container runtimes run with root privileges. If the container runtimes have a vulnerability that allows codes to run with root privileges on the host, malicious people can execute their malicious codes on the host environment with root privileges by exploiting the vulnerability.

Rootless containers improve the container's security by running its runtime without root privileges. If the container runtime has vulnerabilities that allow malicious codes to be executed outside the containers, the codes are executed without root privileges. So that, the damage caused by malicious codes will be less than that of executed in container runtime with root privileges.

In ordinary container runtimes, root privileges are used when they create namespaces like PID namespaces and configure the network including network namespaces. In rootless containers, it uses user namespaces(UserNS) to create namespaces without root privileges. UserNS maps a non-root user like uid=1000 to a fake root user (uid=0). In the host environment, the user is mapped to a non-root user. By using this, container runtimes can perform operations that require root privileges like creating PID namespaces and Mount namespaces.

However, operations related to network interfaces cannot be realized with UserNS alone. This is because that container runtime needs to create a veth pair both in a container and a host. This veth pair is the entrance and exit point for container communication and requires root privileges to create in a host. To achieve container networking without root privileges, rootless containers have networking components called RootlessKit and slirp4netns. Inside a rootless container, there is an intermediate network namespaces(NetNS), a bridge for communication between containers, and a tap device for communication with the outside world. Each container creates additional NetNS within the intermediate NetNS to achieve network isolation similar to that of a normal container. Relaying of communication between the intermediate NetNS and hosts is achieved using RootlessKit and slirp4netns.

2.1.1 RootlessKit's port driver

RootlessKit[8] enables rootless containers including creating namespaces, making /etc writable and managing slirp4netns processes. The port driver is a component contained in RootlessKit to relay communications from outside of an intermediate NetNS to inside of an intermediate NetNS with rootless. When a specific port is to be exposed to the outside world, it has a correspondence between the port in the container and the port on the host to be exposed to the outside. For example, it is used to expose the container's port 80 to the outside world on the host's port 8080. In an ordinary container, this is achieved by combining a veth pair and iptables. However, this is not possible in rootless containers

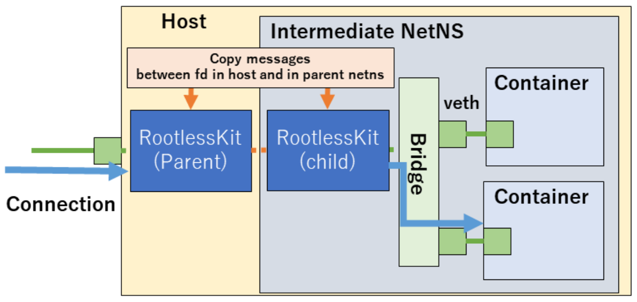

Fig. 1 The overview of RootlessKit's port driver

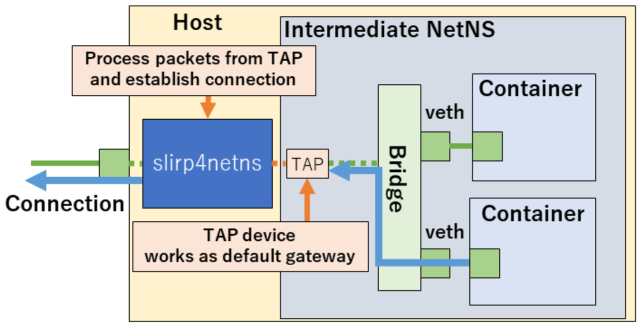
Host
Intermediate NetNS
Process packets from TAP and establish connection
slirp4netns
TAP
Bridge
veth
veth
Container
Container
Connection
TAP device works as default gateway

Fig. 2 The overview of slirp4netns

because the creation of a veth pair between the host and the container requires root privileges. Therefore, RootlessKit provides a method to expose containers' ports without veth pairs.

Fig. 1 shows the overview of RootlessKit's port driver. When port driver exposes a specific port like 8080, a parent daemon is started on the host to listen for the specified port. When the connection is accepted in the parent daemon, the accepted file descriptor(fd) is sent via a Unix domain socket to the child daemon of RootlessKit in the intermediate NetNS containing the container. The child daemon connects to the port of the container. Then, by copying messages between the fd for external connection and the fd for container connection, the child daemon relays messages from the outside to the inside. Since the connection is terminated once in the kernel and only the message is copied in userland, it is faster than a method used in slirp4netns. However, on the container side, the source address of the connection is rewritten to the localhost.

2.1.2 slirp4netns

slirp4netns[9] is a component that provides rootless communication from inside a container to the outside. There are two types of container communication. The first one is inter-container communication. This is already possible through veth pairs and bridges created in the intermediate NetNS. The second one is outbound communication to endpoints outside of the intermediate NetNS. This communication is relayed by slirp4netns.

Fig. 2 shows the overview of slirp4netns. slirp4netns uses the tap device in intermediate NetNS to listen for packets to external endpoints. At this time, the tap device's fd is sent from within the intermediate NetNS to the host. On the host side, slirp4netns receives Ethernet frames and processes them with a userland TCP/IP stack called Slirp[11]. Non-root processes including slirp4netns are not assigned CAP_NET_RAW in terms of security. This is because CAP_NET_RAW allows unprivileged users to sniff and snoop arbitrary packets. So, slirp4netns cannot send the Ethernet frames directly. Slirp reconstructs Ethernet frames and acquires protocols, messages, and the destination of the message. slirp4netns converts messages into compatible syscalls based on them. In more detail, it creates a socket like SOCK_STREAM or SOCK_DGRAM, connects to the external endpoint, and forwards the message. The received message is also processed by slirp, divided into packets, and sent from the tap device, which sends the message to the appropriate container.

Since the communication relay is achieved by processing packets with a tap device, it has the advantage of being compatible with iptables and CNI plugins. However, it has the disadvantage of low performance and increased resource usage because all packets must be processed on the user-land process.

2.2 lxc-user-nic

lxc-user-nic[12] is a component to create veth pair in a container and on a host without root privileges. It creates veth pair as same as veth pair in rootful containers. Since the lxc-user-nic is a setuid-root program, it can be launched by a non-root user to create a veth pair. However, lxc-user-nic effectively runs as the root user. So that, if lxc-user-nic has a vulnerability such as CVE-2017-5985[13] or CVE-2018-6556[14], malicious operations or codes can be executed with root privileges.

While lxc-user-nic provides much better throughput than slirp4netns, lxc-user-nic is rarely used in modern rootless containers due to that security concerns.

2.3 Seccomp Notify

Seccomp(secure computing mode)[15] is a Linux kernel module for controlling syscalls that can be executed by processes. Seccomp can be combined with the container runtime to control syscalls that are executed by processes in a container. Seccomp has supported only static control using predefined policies. With Linux kernel 5.0, dynamic control of syscalls is supported. It is called Seccomp Notify. Seccomp Notify receives information about the syscalls about to be executed via fd, called notify fd. Then, it is possible to dynamically decide whether or not to execute the call based on its arguments.

With Linux kernel 5.9, the function called SECCOMP_IOCTL_NOTIF_ADDFD was added to seccomp. SECCOMP_IOCTL_NOTIF_ADDFD allows the user to install a file descriptor of a supervisor into a target process when a system call is about to be executed. This allows the target process to use the file descriptor that specifies the supervisor's file description or socket.

3. The Design of bypass4netns

We propose bypass4netns, a method to improve communication performance by switching sockets. RootlessKit and slirp4netns relay communications between the host and container NetNS with a daemon or tap device in an intermediate NetNS. bypass4netns bypasses an intermediate NetNS by switching sockets created in the container to sockets created on the host side, without using a tap device.

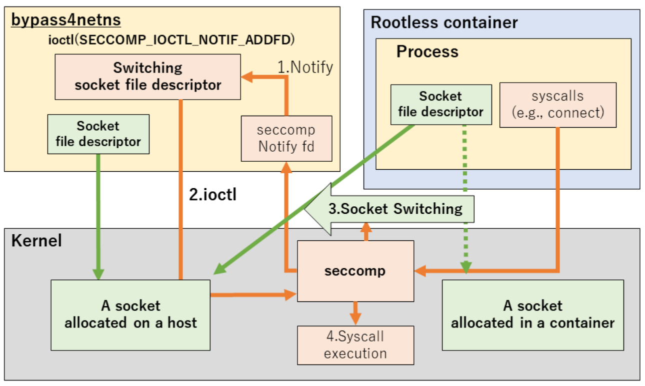

Fig. 3 The overview of bypass4netns

3.1 The Overview of bypass4netns

Fig. 3 shows the overview of bypass4netns. bypass4netns improves communication performance to or from external endpoints by using sockets created on the host rather than sockets created inside the container. External endpoints mean endpoints that exist outside of intermediate NetNS.

bypass4netns works with the rootless container runtime and uses seccomp notify for socket switching. The two major roles of bypass4netns are the creation of sockets on the host and the switching of sockets in the container. bypass4netns waits for seccomp notifications and processes them depending on the kind of syscalls. Inter-container communication is not handled by bypass4netns because it uses veth pairs and bridges in the intermediate netns as before. External communications are handled by bypass4netns. Therefore, bypass4netns handles syscalls that can have a destination address such as connect, sendto, and so on. When the communication is turned out to be to the external endpoint, bypass4netns performs the socket switching.

bypass4netns uses sockets created on the host, and this behavior is similar to an option net=host. An option net=host means that a container uses a NetNS as same as the host. However, using this option results in leaking abstract Unix domain sockets [16]. To ensure the security, bypass4netns does not use the host's NetNS directly and creates sockets selectively on the host. This design also allows assigning unique IP addresses to each of the containers for inter-container communications.

3.2 Socket Switching With Seccomp Notify

As shown in Fig. 3, socket switching is performed with SECCOMP_IOCTL_NOTIF_ADDFD. bypass4netns receives a notify message containing information about the syscall via notify fd. After validation of message IDs, bypass4netns reads the information about the syscall. For syscalls that are not related to socket switching, a response message that allows syscall execution is sent. If a syscall set socket options, all options are recorded for each socket.

When a syscall that determines whether the communication is to a container or an external endpoint is handled, the socket switching is performed as shown in Algorithm 1. When a syscall is handled, bypass4netns creates a new socket. The socket options between the socket in the container and the newly created socket are needed to be in the same state. by-

Algorithm 1 Socket switching procedure

```
1: Read syscall arguments
2: if The syscall is not related to the socket then
3:     Allow to execute the syscall and return
4: end if
5: if The syscall specifies a destination then
6:     if The destination is an external endpoint then
7:         Perform socket switching and return
8:     else
9:         Allow to execute the syscall and return
10:    end if
11: end if
12: if The syscall sets options to the socket then
13:     Record socket option
14: end if
15: Allow to execute the syscall and return
```

pass4netns configures the new socket based on the record of the options corresponding to the handled socket. Then, using SECCOMP_IOCTL_NOTIF_ADDFD, bypass4netns switches the socket pointed by the file descriptor in the argument of the handled syscall to the created socket.

4. Analyzing Socket Behavior

Section 3 describes the design of bypass4netns and the overview of socket switching. In an implementation, the below information is required.

- What kind of syscalls are executed for sockets?
- What are the syscalls that specify the destination of a socket?
- What are the socket-related syscalls that are executed before socket switching?

To implement bypass4netns, these points need to be investigated in advance.

Operations on sockets are performed by syscalls with file descriptors in their arguments. Some syscalls are used for more than just sockets, or some syscalls take file descriptors as arguments but are not used for sockets. Therefore, syscalls for sockets are not explicitly given. We investigated the above points by collecting the syscalls that are performed on sockets in common applications(ping, wget, curl, nginx, apache).

To collect the syscalls, we used a patched tracee[17], which is an eBPF-based syscall tracer. We implemented the analysis script to retrieve the operations on each file descriptor of the sockets from the collected syscalls trace. This script outputs the series of syscall executions on each socket. The target sockets are SOCK_DGRAM or SOCK_STREAM. This script also takes care of fork and clone since file descriptors are replicated between processes by fork and clone,

Table 1 shows the syscalls executed to sockets. sendto(dst) means sendto(2) with destination address and bind(port=0) means bind(2) with port = 0 in its arguments. From Table 1, syscalls that specify the destination address or the port to be published are found. The syscalls are connect, sendto(dst), and bind

We investigated syscalls executed before connect, sendto(dst), and bind. The results are shown in Table 2.

syscalls	ping	wget	apache	nginx	iperf3 (client)	iperf3 (server)
socket	19	32	2	2	2	2
getsockopt	0	0	0	0	5	5
connect	18	30	0	0	2	0
write	0	0	0	0	22551	8
getsockname	9	14	1	0	2	3
getpeername	0	0	0	0	1	2
select	0	0	0	0	4524	56998
fcntl	0	4	8	0	4	2
close	19	32	2	1	2	3
setsockopt	0	0	6	4	1	5
read	0	2	4	0	8	19006
accept	0	0	0	0	0	4
bind	0	0	1	2	0	2
listen	0	0	1	4	0	2
bind(port=0)	1	2	0	0	0	0
sendto(dst)	2	4	0	0	0	0
poll	2	5	0	0	0	0
recvfrom	3	4	0	2	0	0
ioctl	0	2	0	2	0	0
writev	0	2	1	1	0	0
lseek	0	2	0	0	0	0
readv	0	10	0	0	0	0
accept4	0	0	2	2	0	0
epoll_wait	0	0	7	36	0	0
shutdown	0	0	1	0	0	0
sendfile	0	0	0	1	0	0

Table 1 The number of syscalls related to sockets traced from applications

syscalls	ping	wget	apache	nginx	iperf3 (client)	iperf3 (server)
getsockopt	2	0	0	0	0	0
setsockopt	0	4	0	0	4	3
close	0	0	0	0	1	0
ioctl	0	0	0	0	0	2

Table 2 The number of syscalls executed before socket switching

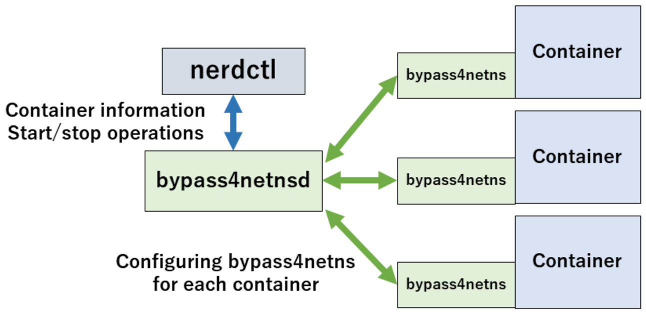

Fig. 4 The implementation of bypass4netns

Syscalls(fcntl, ioctl, setsockopt) which set socket options are executed.

From these results, we can say that syscalls with destination address or binding port are connect, sendto(dst) and bind. fctnl, ioctl,and setsockopt configure socket options and need to be recorded.

5. Implementation

We implemented bypass4netns based on the design described in Section 3 and the analysis in Section 4. The target container runtime is nerdctl[18]. As shown in Fig. 4, we implemented bypass4netns and bypass4netnsd. Also, we implemented some patches in nerdctl to support bypass4netns. The implementation is published at `https://github.com/naoki9911/bypass4netns`. Upstreamed version is available at `https:`

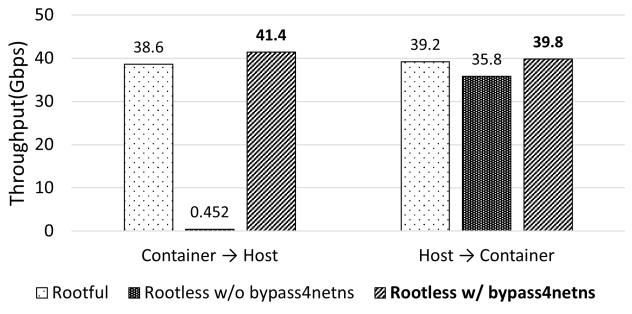

Fig. 5 Throughput in case A

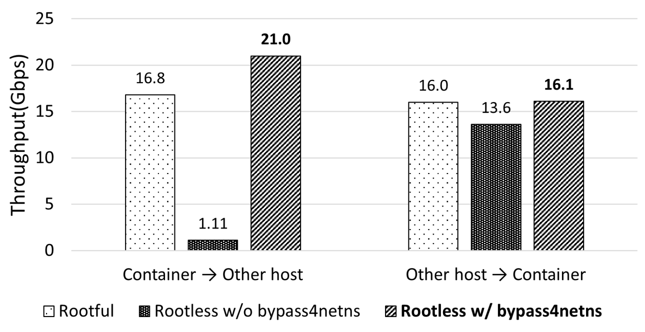

Fig. 6 Throughput in case B

//github.com/rootless-containers/bypass4netns.

bypass4netns

bypass4netns receives seccomp notifications from containers and switches sockets as needed. bypass4netns itself is assigned on a per-container basis.

bypass4netnsd

bypass4netnsd manages bypass4netns assigned to each container. When a container with bypass4netns enabled starts, bypass4netnsd receives information from nerdctl such as the port to be published and the ID of the target container. bypass4netnsd starts bypass4netns based on such information and terminates bypass4netns when a container is stopped.

nerdctl

nerdctl[18] is a Docker-compatible CLI for containerd[19]. it cooperates with containerd and provides container functionality. In rootless containers, nerdctl configures rootless components including RootlessKit and slirp4netns in order when a container starts or stops. When a container with bypass4netns starts, nerdctl notifies information about the container to bypass4netnsd and waits for bypass4netns to start. Also, when the container stops, nerdctl notifies bypass4netnsd and waits for bypass4netns to stop.

6. Performance Evaluation

We evaluated the performance of communications with bypass4netns implementation described in Section 5. The performance evaluation is performed on Hyper-V virtual machines(VMs). The detailed specifications are as follows.

- Host CPU: Ryzen7 PRO 5850U (8Core 16Threads)
- Host Memory: 32GB
- Host OS: Windows 11 Pro (build 22000.739)
- VM assigned CPU: 4 Core
- VM assigned Memory: 8GB
- VM OS: Ubuntu 21.10 (kernel 5.13.0-1025-azure)

6.1 Throughput

We measured throughput using iperf3. The performance was evaluated in two cases, **Case A. throughput between a container and host on the same VM** and **Case B. throughput between a container and different VM**.

Case A. A container and a host on the same VM

The result is shown in Fig. 5. From a container to a host, rootful containers achieved 38.6Gbps, rootless containers without bypass4netns achieved 452Mbps, and rootless containers with bypass4netns achieved 41.4Gbps. From a host to a container, rootful containers achieved 39.2Gbps, rootless containers without bypass4netns achieved 35.8Gbps, and rootless containers with bypass4netns achieved 39.8Gbps.

Case B. A container and a different VM

The result is shown in Fig. 6. From a container to a host, rootful containers achieved 16.8Gbps, rootless containers without bypass4netns achieved 1.11Gbps, and rootless containers with bypass4netns achieved 21.0Gbps. From a host to a container, rootful containers achieved 16.0Gbps, rootless containers without bypass4netns achieved 13.6Gbps, and rootless containers with bypass4netns achieved 16.1Gbps.

In case A, the throughput with rootless containers without bypass4netns is slower than that of case B. This may be because Slirp implementations are less mature than Linux implementations

In both cases, We measured CPU utilization (user + sys) with the rate of iperf3 limited to 1Gbps The measurement results are shown in Fig. 7. CPU utilization exceeded 25% when bypass4netns is not enabled. This is due to packet processing in slirp4netns, which relays communications between hosts and containers. On the other hand, when bypass4netns is enabled, the CPU utilization is low. This is because bypass4netns only switches sockets and the communication itself is handled by the kernel as is in the rootful containers.

In the results of Fig. 5 and Fig. 6, we can see that the bypass4netns is faster than the rootful container. Using ipftrace2[20], which retrieves the functions performed to process

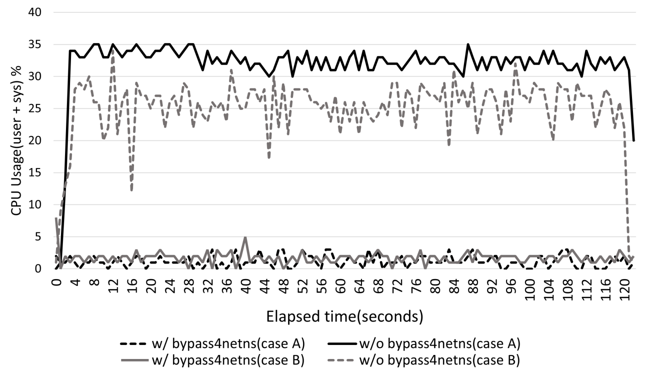

Fig. 7 CPU utilization when doing iperf3 in each case

rootful	rootless without bypass4netns	rootless with bypass4netns
673	922	198

Table 3 The number of the kernel function call to process curl

	without bypass4netns	with bypass4netns
execution time(usec)	3.22	100.05

Table 4 The overhead of syscalls

packets in the kernel, we measured the number of the functions required to process ”curl google.com”. As a remark, the number is sum of the number of kernel function measured in each NetNS. The result is shown in Table 3. Although simple comparisons cannot be made because each kernel function requires a different cost to compute, it can be seen that the number of kernel functions in rootless with bypass4netns is smaller than that with rootful container. The reason for this may be that bypass4netns does not require NAT and other packet processing with iptables or veth pair, which is necessary for the rootful container. As a result, rootless with bypass4netns is considered to be faster and more efficient than the rootful container.

6.2 Syscall Handling Overhead

bypass4netns handles syscalls dynamically using seccomp notify. Therefore, there is a certain overhead in syscall executions. We measured the overhead of syscall execution in bypass4netns by using a program that simply repeats socket, connect, and close. The measurements are perfromed one million times and the means are calculated. The results are shown in Table 4. The results show that the container with bypass4netns takes about 30 times longer to execute the program than a container without bypass4netns. Although syscall overhead exists, the results of the analysis in Section 4 indicate syscalls need to be handled. Therefore, the overhead can be reduced by setting the seccomp profile appropriately. In addition, in real applications, other syscalls are also executed and the number of them is larger than that of socket-related syscalls. So that the impact of overhead is considered to be relatively small.

7. Discussion

7.1 The Combination With Other Components

bypass4netns is a mechanism to improve communication performance by bypassing intermediate NetNS isolation with sockets created on the host. If any communication policies such as iptables policies are set in the intermediate NetNS, they will be ignored. Therefore, it is difficult to control communications using iptables or CNI plugins in intermediate NetNS.

7.2 Security Consideration

External communications with bypass4netns are treated as same as other communications on the host. This is because the sockets switched by bypass4netns are created on the host. Without bypass4netns, as mentioned above, container communication can be controlled within the intermediate NetNS. However, bypass4netns cannot provide access control using iptables or CNI plugins. Therefore, a policy that allows communication from the host and prevents the container cannot be applied. Access control for containers is a future issue.

Seccomp notify has the probability to allow the container to connect to the host’s loopback by exploiting a time-of-check to time-of-use(TOCTOU) attack. This may allow a container to connect to other services hosted on the host. We will address this issue in the future.

7.3 Limitations of bypass4netns

Currently, bypass4netns does not care about VXLAN which is used for an overlay network. Containers are often used with container orchestration tools like Kubernetes. A cluster of Kubernetes is often deployed with overlay networking using VXLAN. However, the configuration of VXLAN interfaces requires root privileges. Usernetes[21] is a Kubernetes distribution based on rootless containers. In Usernetes, by creating VXLAN interfaces in the intermediate NetNS, an overlay network is achieved without root privileges. This causes a problem of slow inter-container communications. However, bypass4netns only supports bypassing of userland sockets. VXLAN interfaces are provided by the Linux kernel and the sockets are created in kernel-land. bypass4netns cannot handle such kernel land sockets and cannot be used in Usernetes to accelerate network communications.

8. Conclusion

In this paper, we propose bypass4netns, a method to improve the TCP/IP communication performance to or from the external endpoints in rootless containers. In a rootless container, the component to relay communications between the intermediate NetNS for the rootless container and the host is the bottleneck. bypass4netns switches the socket in the container to the socket created on the host and bypasses the relay component. We implemented bypass4netns and confirmed that it can achieve communication speeds 10 times faster than conventional rootless containers. Some future issues exist including addressing security concerns and combining with more complicated systems like Usernetes.

References

[1] Docker, Inc.: Docker, https://github.com/docker (2013).
[2] Red Hat, Inc.: Podman, https://github.com/containers/podman (2018).
[3] Kerrisk, M. et al.: namespaces(7), https://man7.org/linux/man-pages/man7/namespaces.7.html (2021).
[4] Kerrisk, M. et al.: capabilities(7), https://man7.org/linux/man-pages/man7/capabilities.7.html (2021).
[5] Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K. and Zhou, Q.: A Measurement Study on Linux Container Security: Attacks and Countermeasures, ACM, p. 418–429 (online), DOI: 10.1145/3274694.3274720 (2018).
[6] NIST: CVE-2019-14271, https://nvd.nist.gov/vuln/detail/CVE-2019-14271 (2019).
[7] Sarai, A. et al.: Rootless Containers, https://rootlesscontaine.rs/ (2017).
[8] Suda, A. et al.: RootlessKit, https://github.com/rootless-containers/rootlesskit (2018).
[9] Suda, A. et al.: slirp4netns, https://github.com/rootless-containers/slirp4netns (2018).
[10] Kerrisk, M. et al.: seccomp_unotify(2), https://man7.org/linux/man-pages/man2/seccomp_unotify.2.html (2021).
[11] Gasparovski, D.: Slirp, https://web.archive.org/web/19970728154029/http://ucnet.canberra.edu.au/slirp/slirp.doc.txt (1995).
[12] Hallyn, S. et al.: lxc-user-nic, https://github.com/lxc/lxc/blob/master/doc/lxc-user-nic.sgml.in (2013).
[13] NIST: CVE-2017-5985, https://nvd.nist.gov/vuln/detail/CVE-2017-5985 (2017).
[14] NIST: CVE-2018-6556, https://nvd.nist.gov/vuln/detail/

CVE-2018-6556 (2018).
[15] Kerrisk, M. et al.: seccomp(2), https://man7.org/linux/man-pages/man2/seccomp.2.html (2021).
[16] Suda, A.: [CVE-2020–15257] Don 't use –net=host . Don 't use spec.hostNetwork ., https://medium.com/nttlabs/dont-use-host-network-namespace-f548aeeef575 (2020).
[17] Shakury, I. et al.: tracee, https://github.com/aquasecurity/tracee (2019).
[18] Suda, A. et al.: nerdctl, https://github.com/containerd/nerdctl/ (2020).
[19] Crosby, M. et al.: containerd, https://github.com/containerd/containerd (2016).
[20] Hayakawa, Y. et al.: ipftrace2, https://github.com/YutaroHayakawa/ipftrace2 (2020).
[21] Suda, A. et al.: Usernetes, https://github.com/rootless-containers/usernetes (2018).