Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Rendering Network Namespaces Obsolete with eBPF

Author:
Lucas Graeff Buhl-Nielsen

Supervisor:
Marios Kogias

Acknowledgements

I would like to thank my supervisor, Marios Kogias, for supporting me throughout the entirety of this project. Marios would always ask the kinds of thought-provoking questions that made me challenge my assumptions, deepen my understanding, and ultimately guide me towards a better solution. I am incredibly grateful for having had the opportunity to benefit from his mentorship.

My four years at Imperial have exceeded my expectations in every way. In particular, I've met some of the most creative, motivated, and inspiring people who I'm grateful to call my friends. Thank you all for the memories, laughter, and lunchtime debates. I cannot wait to continue our adventures together!

Finally, I would like to thank my parents. Aside from being by my side and supporting me in everything I've done, I owe my parents for my love of Computer Science. My parents were always incredibly supportive of my passion of building, and introduced me to the world of software at an early age. I'll never be able to truly express how grateful I am for my parents.

Abstract

State-of-the-art system call rewriting mechanisms are typically implemented in user space and require rewriting binaries, which makes them unsuitable for cloud providers. By extending the capabilities eBPF, this thesis presents an alternative mechanism that allows for kernel space system call argument rewriting. To demonstrate the utility of this mechanism, we use it to implement a lightweight alternative to network namespaces. This allows containers to have an isolated network environment without having to pay the substantial performance penalties associated with creating its own network namespace. This is not feasible with user-space system call interception mechanisms. Evaluation results show that this approach results in 92× reduction in launching 200 containers in parallel and a 60% reduction in the launch time of a single container. Furthermore, this approach improves the throughput of a containerized nginx web server running in an overlay network by 35% and reduces the latency by 33%. The overhead of the eBPF hooks is comparable to current state-of-the-art user-space system call interception mechanisms.

Contents

Chapter 1

Introduction

eBPF is a Linux kernel technology that enables user-defined programs to safely extend and modify kernel behaviour without changing kernel source code. Over the past decade, eBPF has evolved to support a diverse set of use cases, and is now widely adopted in production systems at companies such as Meta [1], Netflix [2], AWS [3], and Cloudflare [4, 5]. In this project, we extend the capabilities of eBPF by introducing support for system call interception, allowing eBPF programs to inspect and modify system call arguments. We leverage this to create a lightweight alternative to network namespaces that provides network isolation for containers without the overhead of creating an entirely separate networking stack. This approach substantially improves container startup time, throughput, and latency compared to traditional namespace-based isolation.

1.1 Motivation

State-of-the-art system call interception mechanisms involve disassembling binaries and rewriting system call instructions such that they instead call a user-space interposition process [6, 7], as shown in Figure 1.1. Cloud providers are often unwilling to rewrite customer binaries because it crosses the boundary established by the shared responsibility model [8] which places the security burden of the application on the customer rather than cloud provider. Furthermore, the performance overhead introduced by user-space interposition is unacceptable for production workloads [9, 10]. Existing kernel-space system call interception mechanisms, such as seccomp [11], kprobes [12] and tracepoints [13], focus on system call filtering and do not allow for safely rewriting system call arguments. This restriction makes these mechanisms unsuitable for many use cases [14, 15]. We believe that eBPF could be extended to provide a system call argument rewriting mechanism that could unlock substantial performance benefits for cloud providers.

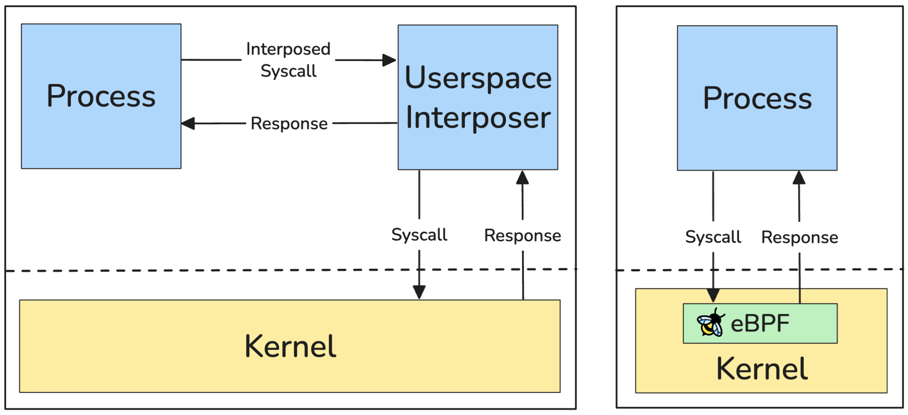


Figure 1.1: A comparison of state-of-the-art user-space interposition (left) and eBPF based system call interception (right).

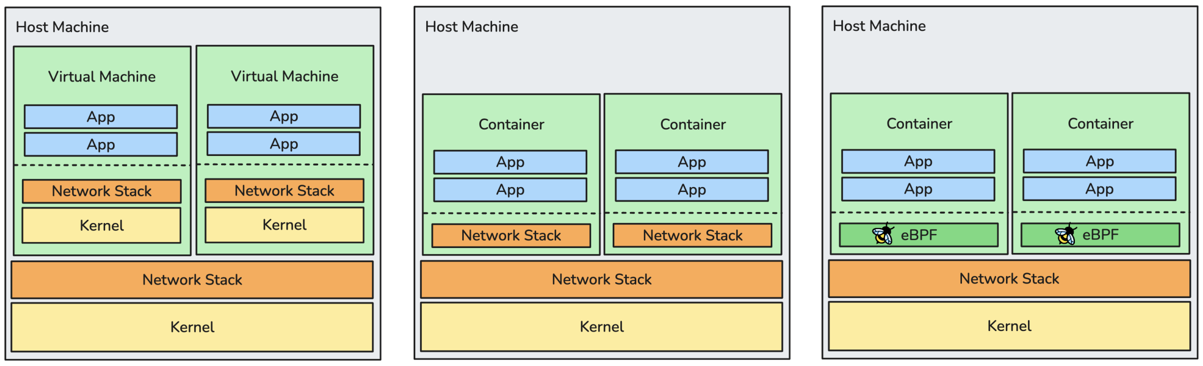


Figure 1.2: A comparison of virtual machines, containers isolated with network namespaces, and containers isolated with an eBPF based system call interception approach.

As a specific motivating example, we consider how such a mechanism could be used to implement a lightweight alternative to network namespaces. Prior studies have demonstrated that the creation and setup of network namespaces is the most significant bottleneck in container startup times [16, 17] and have quantified the impact on throughput and latency in container overlay networks [18, 19]. Standard overlay networks experience a drop in throughput of up to 48% and an increase in latency of up to 85% [18]. With eBPF based system call interception, we can avoid these performance penalties by allowing containers to share the host network stack. This is analogous to how containers avoid the overhead of virtual machines by sharing the host kernel, as shown in Figure 1.2.

1.2 Contributions

In this project, we make the following two contributions:

- Prepare a kernel patch that extends the eBPF subsystem to support safe and exhaustive system call argument rewriting with performance comparable to state-of-the-art user-space system call interception mechanisms
- Design and implement a lightweight alternative to network namespaces that leverages the eBPF system call rewriting mechanism to achieve a 60% decrease in container startup time as well as improving the throughput of container overlay networks by up to 35% while reducing latency by up to 33%.

1.3 Outline

This thesis presents the design and implementation of both the eBPF based system call interception and the specific use-case of a lightweight alternative to network namespaces. We begin by first covering the necessary background information required to understand the contributions (chapter 2 and chapter 3). Following this, we present the design and implementation of a set of eBPF system call hooks that support safe argument rewriting in chapter 4. Then, we describe the design and implementation of a Kubernetes-based lightweight alternative to network namespaces in chapter 5. This design relies on the eBPF system call hooks, and would not be possible without them. In chapter 6, we evaluate the overhead of our eBPF hooks and the performance of our lightweight alternative to network namespaces by comparing it to traditional network namespaces using microbenchmarks and two real-world applications. Finally, we conclude the thesis in chapter 7 and discuss future work.

Chapter 2

Background

In this chapter introduce eBPF as a powerful technology that allows users to extend the functionality of the Linux kernel in a fast and secure way. Then, we briefly cover some existing mechanisms for intercepting, rewriting and filtering system calls. This motivates the need to an eBPF based system call argument rewriting mechanism, which we see as a natural extension to existing eBPF hooks and a push towards a fully extensible kernel. Next, we introduce the technical underpinning of Linux containers and network namespaces as well as the various existing options for container networking. We end the chapter with an introduction to netlink, which is a communication mechanism between userspace processes and the Linux kernel. This chapter provides the necessary context to understand the motivation behind as well as the design and implementation of a new eBPF program that allows for system call argument rewriting and a new network isolation mechanism that offers a more lightweight alternative to network namespaces.

2.1 eBPF

eBPF is a technology that enables its users to extend the functionality of the Linux kernel in a fast and secure way [20]. This is done by injecting custom code into the kernel which can be run without the need to recompile the kernel or load a module. eBPF stands for extended Berkeley Packet Filter. Originally, BPF was a way to write programs that could filter network packets, but has since evolved into a much more powerful mechanism which allows developers to dynamically adjust and extend the behaviour of the Linux kernel. There is now support for a wide range of use cases: it's possible to use eBPF to intercept a selection of network related system calls such as `bind()` and `connect()` and overwrite their arguments.

The big advantage of eBPF is that it allows developers to augment the behavior of the kernel without having to understand or modify the kernel source code. This is in comparison to creating a kernel module or patching the kernel, both of which require a deep understanding of the kernel internals. Furthermore, kernel modules and patches are at risk of crashing the kernel or introducing security vulnerabilities. eBPF programs are verified by the kernel before they are loaded, which means that they are guaranteed to not crash the kernel. Kernel patches can also take months or years to be accepted into the mainline kernel, which introduces a significant overhead for companies by requiring them to maintain a fork of the kernel until the patch is accepted.

eBPF is already gaining significant traction in industry and is being used by large scale organizations such as Meta, Cloudflare, Netflix and Cilium. For example, Meta uses eBPF to implement their load balancer (Katran) [1], Cloudflare uses eBPF to implement their L4 load balancer (Unimog) [4] and their DDoS mitigation system (L4Drop) [5], Netflix uses eBPF for tracing and observability [2] and Cillium uses eBPF to provide more efficient and secure networking for container [21]. This section gives an overview of the key components of eBPF. We start by introducing the concept of eBPF program and attach types as well as maps, then we discuss eBPF helper functions and KFuncs, and end with an explanation of the eBPF verifier.

2.1.1 Program and Attach Types

An eBPF program is a piece of code that can be injected into specific points inside the kernel. Each program has a specific program type, which defines the set of hooks that the program can be attached to. The hook that the program attaches to is known as the attach type. One example of an eBPF program is the `BPF_PROG_TYPE_CGROUP_SOCK_ADDR` program type, which is allowed to attach to multiple hooks in the kernel that are related to socket system calls. For example, it can be attached to the `bind()` system call handler for `AF_INET` sockets using the `BPF_CGROUP_INET4_BIND` attach type.

Programs are called with a context, which is a structure containing information passed from the kernel to the program that can be read or modified by the kernel. For example, the `BPF_PROG_TYPE_CGROUP_SOCK_ADDR` program allows a user to read or modify the IP address and port number that the socket will be bound to. Typically, programs are written in a restricted subset of C and compiled with LLVM to BPF bytecode.

2.1.2 Maps

Maps are a way for eBPF programs to store and retrieve data. They can be shared between multiple eBPF programs, which allows them to communicate with each other. Furthermore, maps can be accessed from user-space, which allows for eBPF programs to communicate with user-space processes. There are multiple different types of maps that offer slightly different functionality:

- `BPF_MAP_TYPE_HASH`: essentially a hash map that has no restrictions on the structure of the key or the value.
- `BPF_MAP_TYPE_ARRAY`: a fixed size array that can be used to store large amounts of data. The key starts at zero, as a normal array.
- `BPF_MAP_TYPE_QUEUE`: a map that resembles a first-in-first-out queue, with the ability to push and pop elements.

Maps can be *pinned* to the filesystem, which means associating the map with a particular file path in the `/sys/fs/bpf` directory. This allows userspace processes and other eBPF programs to share the map and access any of its data.

2.1.3 Helper Functions

Helper functions are functions defined inside the kernel that have been written to be used by eBPF programs. They perform actions that are not possible in standalone eBPF programs, such as reading the PID of the calling process or writing to user space memory. These are actions that would be difficult to verify the safety of, which is why they are implemented as helper functions inside the kernel where their safety can be ensured by kernel developers. Helper functions have a stable API, which means that eBPF programs that use the helper functions are compatible with future kernel releases and eBPF developers have a guarantee that their programs will continue to work as the kernel evolves.

2.1.4 KFuncs

The kernel community is no longer accepting new helper functions, but instead allowing kernel functions to be marked as usable from eBPF programs. The subtle difference is that these functions (called KFuncs) do not necessarily expose a stable API to eBPF programs, so they don't have the same advantages as helper functions. The unsustainable maintenance bloat of eBPF helper functions became too much of a burden for the kernel community.

2.1.5 Verifier

Since eBPF programs are executed directly in the kernel, it's possible that a bug in the eBPF program could cause the kernel to crash. To prevent this, all eBPF programs must be verified before being loaded. This is done by the eBPF verifier, which primarily ensures that the eBPF program does not make any out of bounds or unsafe memory accesses and that the program terminates. The verifier exhaustively analyses every possible execution path of the eBPF program. It does this by placing bounds of the values of all registers (according to the type of the value held) and exhaustively trying every single combination to ensure that it terminates.

2.2 System Call Interception

System call interception is a technique that allows users to monitor, modify, or restrict the behaviour of user-space applications by acting as a man-in-the-middle between the application and the kernel. A distinction can be made between system call interception mechanisms that allow for rewriting arguments and those that only allow for filtering system calls. The former is commonly used in applications like Wine [14] which allows Windows binaries to run on Unix operating systems. The latter is commonly used in security contexts, such as seccomp [11] which allows users to restrict the set of system calls that an application can make. The focus of this section is on system call interception mechanisms that allow for rewriting system call arguments. This section first introduces two user-space interception mechanisms: `LD_PRELOAD` and binary rewriting. It then introduces an eBPF program type that allows for intercepting and rewriting a limited set of socket related system calls.

2.2.1 Loading a Shared Library

Unix operating systems allow for users to load dynamically linked shared library objects into an existing application. Any functions or symbols defined in this shared library will override any identical functions or symbols defined in the original application binary. This can be used to override any `libc` system call wrapper functions. If the shared library exposes the same interface as a `libc` system call wrapper function, then the shared library function will be called instead of `libc`.

```
$ LD_PRELOAD=/path/to/mywrapper.so ./target_program
```

Listing 2.1: An example of using `LD_PRELOAD` to dynamically link a shared library when running an application.

This is suitable for applications that are dynamically linked, but has no effect for statically linked binaries. The shared library is able to perform any actions it likes: it can keep a record of the system call, rewrite the arguments, block the system call or even invoke an entirely different system call. However, an important limitation is that this is not an exhaustive system call interception mechanism. For example, an application can directly invoke system calls rather than using the `libc` wrapper functions. In this case, the shared library will not be called. This makes the approach completely unsuitable for any security applications, as it fails to provide robust system call filtering. Furthermore, for applications like Wine [14] which rely on system call rewriting to port binaries between operating systems, the lack of exhaustiveness is too significant a limitation as many applications will be incompatible.

2.2.2 Binary Rewriting

Current state-of-the-art system call argument rewriting mechanisms [6, 7] rely on disassembling application binaries and rewriting the system call instructions to point to a custom interception function. This approach solves many of the problems with the LD_PRELOAD approach: it is able to exhaustively intercept all system calls, it's efficient, and it has no limitations in what actions the interception function can perform. Such approaches are even able to rewrite system calls made by JIT compilers [6], which has traditionally been a difficult problem to solve [7]. Typically, these mechanisms use the LD_PRELOAD environment variable to load custom shared libraries that contain custom system call handler functions. This makes the interception an extremely lightweight operation. However, this approach is still unsuitable for security applications as a malicious application could attempt to modify or remove the interception code. Fundamentally, any approach which fails to isolate the *intercepted* code from the *interception* code is unsuitable for security applications as it is vulnerable to tampering by the application. Furthermore, cloud providers are unwilling to perform binary rewriting on customer applications. This is because it breaks the boundary established by the shared responsibility model [8], which states that the cloud provider is responsible for security *of* the cloud and the customer is responsible for security *in* the cloud. By rewriting customer binaries, cloud providers are taking on the responsibility of ensuring that the application is secure and that the interception code does not introduce any security vulnerabilities into the application. Furthermore, maintaining the mechanisms for rewriting binaries is a significant burden due to the number of cloud products offered and different image configurations. Despite this, there are various use cases where cloud providers could benefit from system call rewriting. This requires the ability to rewrite system call arguments in kernel space, which can be done with the eBPF socket address hooks.

2.2.3 eBPF Socket Address Hooks

The BPF_PROG_TYPE_CGROUP_SOCK_ADDR eBPF program type offers a set of hooks that allow us to intercept a limited set of socket related system calls [22]. These hooks allow us to both modify the system call arguments and even conditionally block the system call entirely. Furthermore, they provide a set of helper functions that allow a limited set of other system calls to be called from within the eBPF program. For example, you can invoke the bpf_bind() helper from the BPF_CGROUP_INET4_CONNECT hook to force the socket to bind to a particular interface before it attempts to connect to a remote host. Currently, there exist hooks for the following socket related system calls:

- bind() and connect() for INET4 and INET6
- sendmsg() and recvmsg() for UDP (INET4 and INET6)
- getpeername() and getsockname() for INET4 and INET6

```
SEC("cgroup/bind4")
int bind_v4_prog(struct bpf_sock_addr *ctx)
{
    // Block the syscall if the requested IP address is not 10.0.0.1
    if (bpf_ntohl(ctx->user_ip4) != 0x0A000001)
      return 0;

    // Allow the syscall to proceed
    return 1;
}
```

Listing 2.2: An eBPF program that only allows a socket to be bound to the network interfaces with the IP address 10.0.0.1

As a concrete use case, cloud providers can use these hooks to implement a Network Address Translation, or to force all processes inside a cgroup to use a single IP address on a host which has multiple IPs configured [23]. Whilst these eBPF hooks are incredibly powerful, they are limited to `AF_INET` and `AF_INET6` sockets. Currently, eBPF offers no mechanism for intercepting other system calls. Furthermore, the placement of these hooks are in the address family specific system call handlers. This limits the power of the hooks: for example, they are unable to change the address family. Other system call interception mechanisms could be used to downgrade all `AF_INET6` sockets to `AF_INET` sockets, but these hooks would not be able to do that as the address family in the context structure is read only. Cloud providers are much more willing to use these hooks as they don't involve tampering with customer binaries and therefore doesn't break the shared responsibility model [8], as the security of the kernel is already the responsibility of the cloud provider.

2.3 Containers

Over the past two decades, containers have emerged as the standard way [24] to deploy applications, such as web servers [25], databases [26] and in-memory key-value stores [27]. The primary two benefits of containers is that they are lightweight and provide a strong level of isolation between applications, which is useful in the context of managing conflicting application dependencies as well as in a security context [28]. In this section, we first provide the historical context that motivated the need for containers, and then we dive into the fundamental technical components of containers in Linux.

2.3.1 Historical Context

In the early days of computing, applications were run directly on physical machines [28]. An issue that quickly arises with this approach is that applications can often have conflicting dependencies. This means that those two applications cannot be run on the same machine at the same time. Another issues with this approach is that there are no security boundaries isolating applications from one another: if one application is either compromised or malicious, it can potentially compromise all other applications running on the same machine. Yet another issue is that there was no way to restrict the amount of resources that a single application could consume, which meant that one application could starve others of resources such as CPU or memory and lead to a significant degradation in performance. Again, this could be the result of a malicious, compromised or malfunctioning application.

These issues motivated the deployment of applications on virtual machines. A virtual machine is an emulation of a physical machine that is able to run all the same components as a physical machine (such as an entire operating system) [28]. Virtual machines allowed for the complete isolation of applications from one another, meaning that one compromised or malicious application could not affect other applications running on the same physical host. However, virtual machines are an extremely heavy-weight solution to deploying an application since they require that a full operating system be deployed alongside it. The overhead of running a full operating system for each application results in a substantial decrease in the number of applications that can be deployed on a single physical host. This meant that virtual machines were not a particularly cost-effective solution for deploying applications.

As a result, containers were developed as a light-weight approach of achieving isolation between applications. A container is a group of processes that have their own view of the host kernel: they have their own set of process IDs, networking stack, hostname and filesystem. Furthermore, it's possible to place constraints on how many resources processes belonging to a container can consume. The processes inside a container run on the host kernel, which means that they do not require a full operating system to be deployed alongside them. This means that many more applications can be run inside containers on a single physical host than virtual machines. Containers also provide the foundations of serverless cloud functions [29], which are short-lived applications that are typically used to run small pieces of code in response to some event. Containers are appropriate for this use case because of their very short startup time in comparison to virtual machines.

2.3.2 Implementation Overview

The historical context of containers provides us with the perspective required to understand the technical implementation of containers in Linux. As mentioned, containers are a way of placing processes into groups such that those processes have an isolated view of the host kernel and are subject to resource constraints. To achieve this, Linux provides two key components: namespaces and control groups (cgroups). Namespaces provide isolation in terms of what a process can observe about the state of the system. Control groups allow you to place restrictions on resource consumption. In the following two subsections, we explain each of these components in more detail.

Namespaces

Linux offers eight different types of namespaces [30], each of which provide a different aspect of isolation. The PID namespaces isolates the process ID number space [31]. By default, every process in Linux is assigned a unique process ID (PID) that is used to identify it. You can view the PIDs of processes running on the system by running the `ps -A` command:

```
$ ps -A
    PID TTY          TIME CMD
      1 ?        00:01:41 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 rcu_gp
      ...
2064359 pts/4    00:00:00 python3
2064360 pts/3    00:00:00 bash
2064377 pts/3    00:00:00 ps
```

Since the PID namespaces are hierarchical, all processes running on the system are visible when running `ps -A` in the host PID namespace. For example, if we create a new PID namespace and start a python process in it, we'd be able to see that process by running `ps -A` in the host PID namespace. This process may be assigned a PID of 2064579 in the host PID namespace, but would be assigned a much lower PID in our new PID namespace (e.g 3). The python process would not be visible from other PID namespaces that are not a parent of the PID namespace in which it is running. This is an important for containers: we don't want to allow applications inside containers to see what other processes are being run on the system (potentially by other containers belong to other users). Not all namespaces are hierarchical. For example, network namespaces all exist alongside each other and create entirely separate instances of the Linux network stack.

As mentioned, namespaces allow you to control what a process can observe about the state of the system. There are several different types of namespaces in Linux, each of which controls a different aspect of the system.

To create a new namespace, you can use the `unshare()` syscall. This syscall takes a set of flags that specify which namespaces you want to create. It also takes a program argument, which is the program that will be run inside the new namespaces.

```
lucas@machine:~$ sudo unshare --mount --uts --ipc --net --pid --user --cgroup
    ↪ --time --fork --mount-proc bash
nobody@machine:/home/user$ ps -A
    PID TTY          TIME CMD
      1 pts/4    00:00:00 bash
      7 pts/4    00:00:00 ps
nobody@machine:/home/user$ ip l
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen
    ↪ 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
nobody@machine:/home/user$ ls
ls: cannot open directory '.': Permission denied
```

Listing 2.3: An example of how the `unshare()` system call can be used to place processes in new namespaces

Listing 2.3 shows the `unshare()` syscall in action: the PID numbers have been reset to 1, the namespaces isolate the network interfaces so that only the loopback interface is visible, and the user namespace boots us out of the 'lucas' user and into the 'nobody' user, which means that we have no permissions to read the current directory.

Control Groups

Whilst namespaces provide isolation in terms of what a process can observe about the state of the system, control groups allow processes to be placed in hierarchical groups that can be used to restrict the amount of resources that they can consume [32].

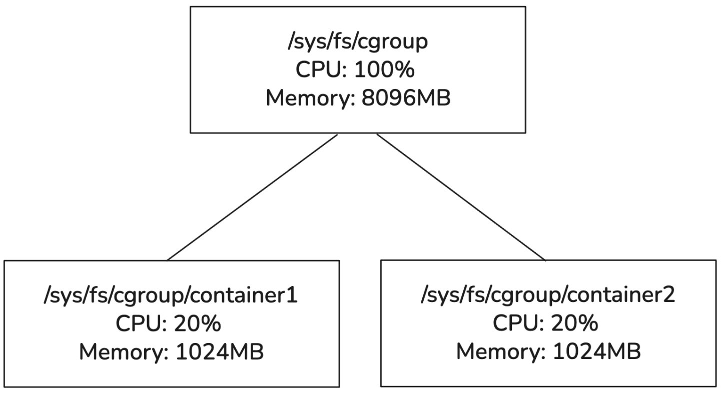


Figure 2.1: The cgroup hierarchy in Linux: new cgroups can be created by creating subdirectories under `/sys/fs/cgroup`. Files within these subdirectories control various resource limits such as CPU and memory.

A cgroup is created by creating a new directory under `/sys/fs/cgroup`, and processes can be placed in a cgroup by writing their PIDs to the `cgroup.procs` file in that directory. All child processes are placed in the same cgroup as their parent process. At its core, a container is created by placing the program that the `unshare()` syscall executes in a cgroup, and then setting resource limits on that cgroup (by modifying files within the cgroup directory, as shown in Figure 2.1).

2.4 Network Namespaces

This section explains how network namespaces are used in containers to provide network isolation [33] inside a container. Network isolation is necessary to prevent a faulty, compromised or malicious process inside a container from viewing or manipulating network state controlled by other processes outside the container. For example, a process inside a container should not be able to see that another container is establishing a TCP connection to a remote host and should not be able to inspect or modify the packets being sent over that connection.

The network isolation provided by network namespaces can be seen by running a `bash` shell inside a new network namespace, as shown in Listing 2.4. Processes inside the new network namespace don't have access to any of the network interfaces that are present in the host's network namespace, which means that they cannot communicate with any remote hosts.

```
lucas@machine:~$ sudo unshare --net bash
root@machine:/home/lucas % ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state UP group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
root@machine:/home/lucas $ ping 8.8.8.8
ping: connect: Network is unreachable
root@machine:/home/lucas curl google.com
curl: (6) Could not resolve host: google.com
root@machine:/home/lucas $ ping -c 1 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.027 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.027/0.027/0.027/0.000 ms
```

Listing 2.4: When a new network namespace is created, by default processes inside that namespace cannot access any remote hosts and only have access to the loopback interface.

Providing network isolation by preventing processes from accessing the networking functionality of the kernel is rather limiting: containers very often need to be able to communicate with the outside world in order to function. To support this, Linux provides a powerful set of virtual network devices that can be placed inside a network namespace such that processes can use them to communicate over the host network.

2.4.1 Virtual Network Devices

We can allow processes inside a network namespace to communicate with a remote host by using Linux's virtual network devices. In particular, the virtual ethernet device (veth) and the virtual bridge. There are many other types of virtual network devices, such as tap devices, but we will only focus on veths and bridges in this section.

A virtual ethernet device is often compared to an ethernet cable: packets transmitted on one end of the veth device will immediately be received on the peer end of the veth device [34]. Each end of the veth device can be placed in a different network namespace. By doing placing one end of a veth inside a network namespace and the other end in the host's network namespace, we can allow processes inside the network namespace to send data to processes running inside the host network namespace.

By attaching the host end of the veth device to a bridge [35], we can forward the packets to the host network interface that is connected to the outside world. This requires the use of something like network address translation (NAT), which rewrites the source IP address to that of the host network interface so that the remote host knows how to send packets back. Figure 2.2 shows the setup of this.

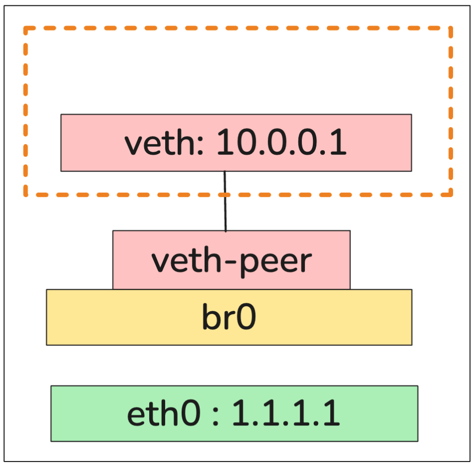


Figure 2.2: A basic setup of a network namespace in Linux which uses a virtual ethernet device and a bridge to communicate with remote hosts. The orange dotted line represents a network namespace.

```
lucas@machine:~$ ip netns exec myns tcpdump -i veth
listening on veth
14:56:10 IP 10.0.0.1 > 8.8.8.8: ICMP echo request
14:56:10 IP 8.8.8.8 > 10.0.0.1: ICMP echo reply
---
lucas@machine:~$ tcpdump -i veth-peer
listening on veth-peer
14:58:13 IP 10.0.0.1 > dns.google: ICMP echo request
14:58:13 IP dns.google > 10.0.0.1: ICMP echo reply
---
lucas@machine:~$ tcpdump -i br0
listening on br0
14:59:59 IP 10.0.0.1 > dns.google: ICMP echo request
14:59:59 IP dns.google > 10.0.0.1: ICMP echo reply
---
lucas@machine:~$ tcpdump -i eth0
listening on eth0
15:01:53 IP 1.1.1.1 > dns.google: ICMP echo request
15:01:53 IP dns.google > 1.1.1.1: ICMP echo reply
```

Listing 2.5: `tcpdump` allows us to follow the journey of an ICMP ping packet sent from the `myns` network namespace through all of the network interfaces

Listing 2.5 shows the output of `tcpdump` when run on the veth device inside the network namespace, the peer veth device in the host network namespace, the bridge and the host network interface. This shows the NAT masquerading in action: the source IP address of the ICMP echo request is rewritten to that of the host network interface. The reply is then forwarded back up the same path.

2.4.2 Kernel Implementation

To further understand how network namespaces provide network isolation between processes, we can look at their high level implementation details inside the Linux kernel. Network namespaces are represented by the `struct net` type. Each process, which is internally represented by the `struct task_struct` type, has a pointer to a `struct net` object. When a process is forked, a new `struct task_struct` object is created, and it inherits the `struct net` pointer from its parent process. This means that it inherits the same network namespace as its parent process. The `unshare()` system call behaves slightly differently: it allocates a new `struct net` object and assigns it to the process that is being created.

The `struct net` object contains fields that represent network state. For example, it contains a hash map containing all the network devices belonging to that network namespace. These network devices are represented by the `struct net_device` type. We also store the routing table, forwarding rules and socket state in the `struct net` object. In fact, all state that makes up the Linux network stack is stored in the `struct net` object. One important detail is that a `struct net_device` object cannot be shared between multiple `struct net` objects. This restriction is foundational to how network namespaces provide network isolation. By enforcing this restriction, the kernel can reuse the same data structures to track entirely isolated network state, such as routing tables and socket state, without having to worry about which processes should or should not be able to see or manipulate that state. Network isolation is a natural by-product of the ability to create entirely separate network stacks and assign them to processes. Allowing network namespaces to belong to multiple `struct net` objects would make the job of the kernel much more difficult and the implementation required to support all networking functionality would be much more complex.

2.4.3 Overhead

Linux provides an elegant and simple way to provide network isolation by creating and maintaining entirely separate network stacks and providing virtual network devices. However, this comes at a cost. The first cost is due to the fact that network devices cannot be shared between network namespaces. If we want processes inside a network namespace to be able to communicate with a remote host, we must either use virtual network devices (as shown in Figure 2.2) or move a physical network device to the namespace. If we choose the first option, we create additional communication overhead as packets must traverse the Linux network stack twice: once in the network namespace and once in the host network namespace. If we choose the second option, then we cannot use that physical network device in any other network namespace.

Furthermore, as analysed by [16] and discussed in section 3.3, the second cost is that moving network devices between network namespaces requires holding a lock (to ensure that the network device can only belong to one network namespace at a time). This is an expensive operation and can result in serious contention when attempting to create many network namespaces in parallel. This sort of workload is not an uncommon pattern. For example, serverless cloud functions [29] will often create an entire network namespace for each function invocation despite the fact that the function may only run for a few milliseconds. This lock contention can lead to the namespace creation time being substantially longer than the application run time [16].

This overhead is analogous to the original motivation for containers: virtual machines require an entire operating system to be deployed alongside each application. The overhead of doing this is so substantial that organizations looked for a more lightweight solution, resulting in the development of containers. Similarly, containers require that an entire networking stack be created for each application. The aim of this thesis is to determine whether we can eliminate the need for creating an entire networking stack to provide network isolation.

2.5 Container Networking

Network namespaces and virtual network devices provide developers with an extremely flexibility mechanism for providing networking functionality to containers. The example outlined in Figure 2.2 is only one of many possible setups that can be used to provide networking capabilities to containers. This section gives a brief overview of some of the most common container networking configurations, and then provides a more focused look at the VXLAN overlay container networking approach, the overhead of which is a core focus of this thesis. It's useful to separate container networking into two categories: intra-host and inter-host. Intra-host networking refers to networking between containers running on the same host, while inter-host networking refers to networking between containers running on different hosts.

2.5.1 Intra-host

There are four common networking modes for intra-host container communication: none, bridge, container and host. We briefly explain each of these in the following subsections. Figure 2.3 shows a diagram of these four modes.

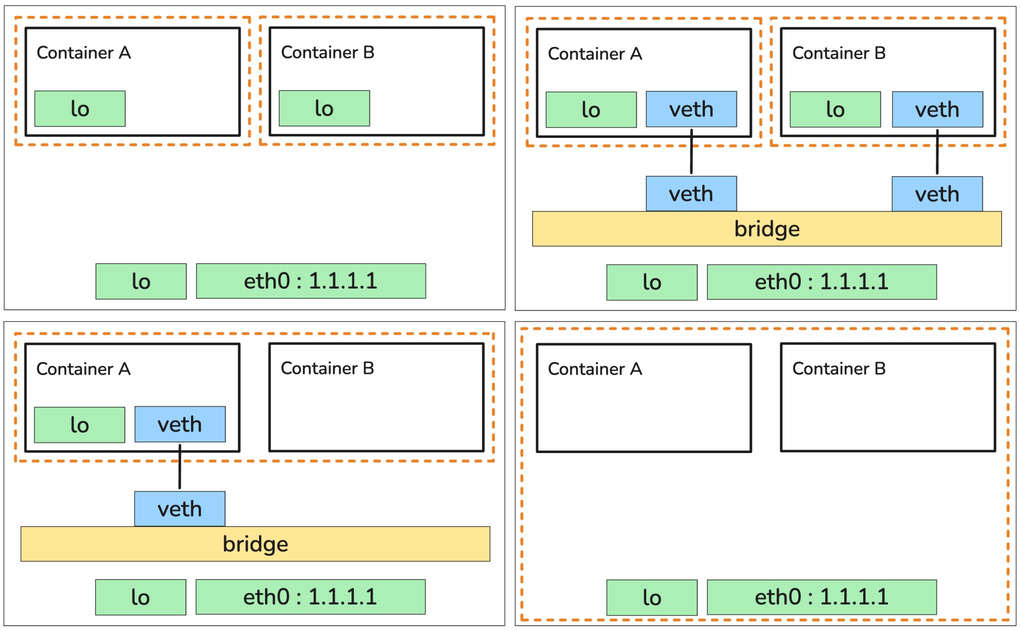


Figure 2.3: The four common intra-host container networking modes. Top left: none, top right: bridge, bottom left: container and bottom right: host. The orange dotted line represents a network namespace.

None

The `none` networking mode is used when a container does not require any networking capabilities. In this mode, the container is only assigned a loopback network interface which allows processes inside the container to communicate with other processes inside the same container. An example of when this might be useful is when you want to run untrusted code. By explicitly disabling any network capabilities you can ensure that the code cannot access the network.

Bridge

The `bridge` networking mode creates a Linux bridge along with a virtual ethernet device. One end of the virtual ethernet device is placed inside the container's network namespace, while the other end is placed in the host's network namespace and attached to the bridge. This places containers in a virtual LAN so that they can communicate with each other, but not the outside world.

Container

The `container` networking mode shares a network namespace among multiple containers. This is the default setup for Kubernetes pods, where the namespace is shared among all containers in the same pod. This allows a greater level of flexibility by allowing containers within the same pod to communicate using the loopback device and address family types such as `AF_UNIX`, which are optimized for intra-host communication. This is appropriate when all containers belong to a single user and are therefore trusted. The drawback of this approach is that it offers no network isolation between containers, which may make it unsuitable for running untrusted code.

Host

Finally, the `host` networking mode places the container in the host's network namespace. This networking mode offers no network isolation between the container and the host, and is therefore unsuitable for running untrusted code. However, it offers the best performance in terms of throughput, latency and container startup time as it does not require the overhead of creating or using network namespaces and virtual network devices.

2.5.2 Inter-host

Two of the most common approaches to inter-host container networking are NAT and overlay. Host mode can also be used for inter-host networking, but we won't discuss it here because we've already discussed it in the intra-host setting, and it isn't particularly common in practice.

NAT

`NAT` mode maps a `<host-ip>:<host-port>` pair to a `<container-ip>:<container-port>` pair. Whenever a packet is received on the host network interface, the kernel routes it to the container by rewriting the destination IP address and port to that of the container. The reverse transformation happens when a packet is sent from the container. `NAT` doesn't tend to scale well because it requires keeping track of a large amount of per connection state and limits the number of source ports that an application can use.

Overlay

The `overlay` networking mode allows containers to communicate with each other as if they were on the same local area network (LAN) even if they are running on different hosts. This is similar to the `bridge` intra-host networking mode but extended to multiple hosts. Overlay networks usually involve the encapsulation of container packets in another set of UDP headers which are then sent over the host network interface. This creates a 'tunnel' between two hosts so that the containers think they are on the same LAN. The trade-off of overlay networks is that they require additional processing to encapsulate and decapsulate packets, which can lead to increased latency and reduced throughput. Furthermore, the encapsulation reduces the maximum transmission unit (MTU) of the network, which can lead to fragmentation and further reduce throughput. In the next section, we take a deeper look at the VXLAN overlay network implementation, which is one of the most common overlay networks.

2.5.3 Virtual Extensible Local Area Network

VXLAN is the technology on which many overlay container networking mode implementations [36] rely on. It can be understood as a tunnelling scheme to overlay Layer 2 networks on top of Layer 3 networks [37]. This section will provide a brief explanation of what this means and how it works, specifically in the context of inter-host container networking.

The fundamental aim is for containers to be assigned a static private IP address that is independent of the underlying machine that it is running on. This way, containers can communicate with one another as if they were on the same LAN, despite actually being connected via the internet. Figure 2.4 demonstrates this idea, with the virtual LAN being the 'overlay' network and the underlying internet being the 'underlay' network.

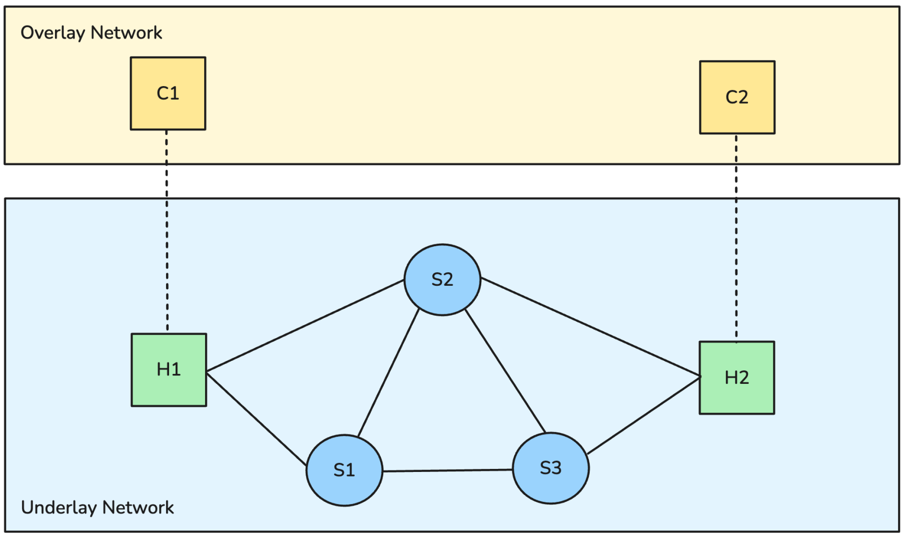


Figure 2.4: An overlay network running on top of an underlay network

A VXLAN tunnel endpoint (known as a VTEP) is an interface which implements the VXLAN protocol, and is therefore responsible with encapsulating all outgoing packets and decapsulating all incoming packets.

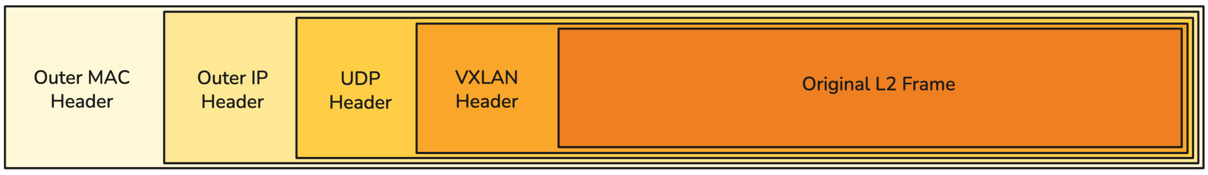


Figure 2.5: VXLAN header encapsulation

When outgoing packets are encapsulated, we specify the source and destination IP addresses and port numbers of the machines that the containers are running on. This way, we can use the underlay network to deliver the packets (via UDP) to the correct VTEP, which can then decapsulate the packet and route it to the correct container. Each VTEP is responsible for maintaining a mapping between the MAC address of each node on the overlay network and the public IP address of the machine that hosts the overlay node. This mapping is used to fill in the destination IP address in the UDP header during encapsulation. There are different mechanisms for a VTEP to learn these mappings, but most commonly used in container networking solutions like Docker Swarm is an unicast-based mechanism which takes advantage of a control plane. The controller is a node which stores the mappings in a key-value store and is statically assigned such that all other nodes only need to know its underlying IP address in order to find the underlying IP address of all other nodes.

Traditional VXLAN-based container overlay networks introduce a communication overhead due to the fact that packets have to traverse the kernel networking stack twice (for an egress packet, once in the host namespace and then again in the container namespace once the VTEP has removed the encapsulation headers). This has been shown to have a significant impact on the latency and throughput of containerized applications compared to applications running directly in the host network namespace [18].

2.6 Netlink

Netlink is a means for user-space processes to communicate with the Linux kernel [38]. User-space processes can use netlink with the standard POSIX socket interface to retrieve information about the state of the kernel. The protocol used when creating the socket determines the type of information that the user-space process can retrieve. For example, three of the supported protocols are:

- `NETLINK_ROUTE`: information about routing and network devices.
- `NETLINK_SOCK_DIAG`: information about currently open sockets and their state.
- `NETLINK_FIB_LOOKUP`: information about the forwarding information base.

```
struct nlmsghdr {
    __u32 nlmsg_len;    /* Size of message including header */
    __u16 nlmsg_type;   /* Type of message content */
    __u16 nlmsg_flags;  /* Additional flags */
    __u32 nlmsg_seq;    /* Sequence number */
    __u32 nlmsg_pid;    /* Sender port ID */
};
```

Listing 2.6: The netlink message header. The `nlmsg_type` tells the kernel what information we want to receive (e.g a list of network interfaces), the `nlmsg_pid` is a unique port identifier for that socket so that the kernel can correctly route the information to the socket receive buffer.

```
int sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

struct {
    struct nlmsghdr nh;
    struct ifinfomsg ifm;
} req;

memset(&req, 0, sizeof(req));
req.nh.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg));
req.nh.nlmsg_type = RTM_GETLINK;
req.nh.nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
req.nh.nlmsg_seq = 1;
req.ifm.ifi_family = AF_UNSPEC;

struct sockaddr_nl addr = {.nl_family = AF_NETLINK};

bind(sock, (struct sockaddr*)&addr, sizeof(addr));

sendto(sock, &req, req.nh.nlmsg_len, 0, NULL, 0);

...
```

Listing 2.7: A client that sends a `RTM_GETLINK` message to the kernel so that it can receive information about the network interfaces in its network namespace.

In order to enforce network isolation between processes, we must have the ability to control netlink messages that are sent by the kernel as well as the actions that are taken by the kernel in response to user-space netlink messages. This is because netlink is used to retrieve information about the state of the network, such as the list of network interfaces. For example, the `ip link` command creates a `NETLINK_ROUTE` socket and sends a netlink message to the kernel requesting a list of network interfaces using the `RTM_GETLINK` message type. The kernel then responds with a list of network interfaces in the network namespace corresponding to the calling process.

Chapter 3

Related Work

Prior work has explored various approaches to address the overheads of network namespaces in overlay networks. Slim and ONCache focus on reducing the communication overheads, while Particle aims to increase the parallelism of launching several hundreds or thousands of containers at once.

3.1 Slim

Slim is a container overlay network that provides a translation layer that sits between the container and the host network stack, allowing containers to achieve near-native network performance.

3.1.1 Quantifying VXLAN Overhead

The authors of Slim quantify the overhead of the double encapsulation and double network stack traversal in a VXLAN-based overlay network. They compare the throughput and latency of a VXLAN-based overlay network to the throughput and latency achieved between two hosts on the same physical network.

Setup	**Throughput (Gbps)**	**RTT (μs)**
Intra, Host	48.4 ± 0.7	5.9 ± 0.2
Intra, Overlay	37.4 ± 0.8	7.9 ± 0.2
Inter, Host	26.8 ± 0.1	11.3 ± 0.2
Inter, Overlay	14.0 ± 0.4	20.9 ± 0.3

Table 3.1: Throughput and latency overhead of overlay networks [18]

Table 3.1 shows that the overhead of the overlay network is substantial, with a 23% reduction in throughput and a 34% increase in round-trip time (RTT) for intra-host communication, and a 48% reduction in throughput and an 85% increase in RTT for inter-host communication.

3.1.2 Design and Implementation

Slim consists of three components: SlimSocket, SlimRouter and SlimKernModule. SlimSocket is a user-space shim-layer that exposes the POSIX socket API to intercept socket related system calls such as bind and connect. SlimSocket passes the system calls to SlimRouter, which is a process that runs in the host network namespace. SlimRouter then executes the system calls inside the host network and passes the file descriptors back to the container via SlimSocket. SlimKernModule is an optional kernel module that can be used to restrict the operations that the container can perform on the host network socket file descriptors.

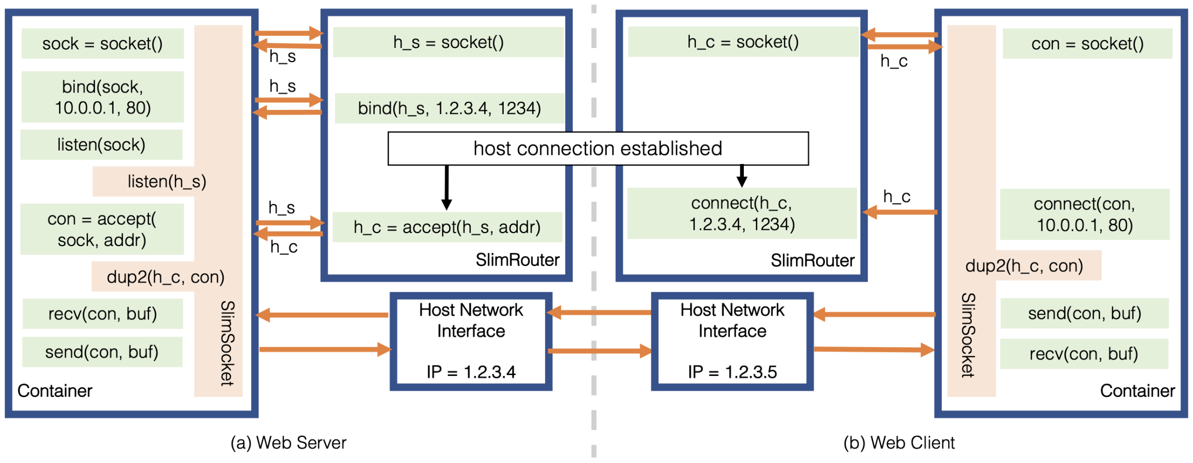


Figure 3.1: TCP connection setup between a web client and server atop Slim [18].

SlimRouter dynamically translates the container IP address and port number to the host IP address and any free port on the host, and distributes this mapping among the entire cluster so that clients can connect to the server. One important caveat of Slim is that it sets up two TCP connections: the first TCP connection uses the standard VXLAN overlay network, and tells the server which host IP address and port the client is about to make a connection from. This allows the server to perform the same translation as the client when it receives a request.

As a result, Slim can achieve throughputs and latencies akin to that of running applications directly on the host, while still maintaining the same isolation guarantees as VXLAN-based overlay networks.

3.1.3 Limitations

Whilst Slim achieves near-native throughput and latency for TCP, there are a a few limitations. We summarize three of them below.

- **UDP**: Slim does not support UDP, ICMP or other non-connection oriented protocols. The authors argue that this is an acceptable trade-off due to the fact that UDP is used when the connection setup time of TCP is too high, and Slim increases the connection setup time meaning that it isn't appropriate for UDP anyway.
- **Connection Setup Time**: Slim requires two TCP connections to be established between the client and server, which more than doubles the time it takes to establish a connection.
- **Container Launch Time**: Slim still requires the container to create a standard VXLAN overlay network, which means that there is no performance gain when launching containers.

3.2 ONCache

ONCache is a cache-based overlay network that eliminates the extra overhead of VXLAN-based overlay networks while maintaining flexibility and compatibility. It is implemented with eBPF tc (traffic control) programs. We specifically focus on the ONCache-t design which proposes a slight modification to the VXLAN overlay protocol to remove the overhead of double encapsulation.

3.2.1 Design and Implementation

The authors design and implement four tc eBPF programs: the egress cache init, ingress cache init, the egress fast path and the ingress fast path.

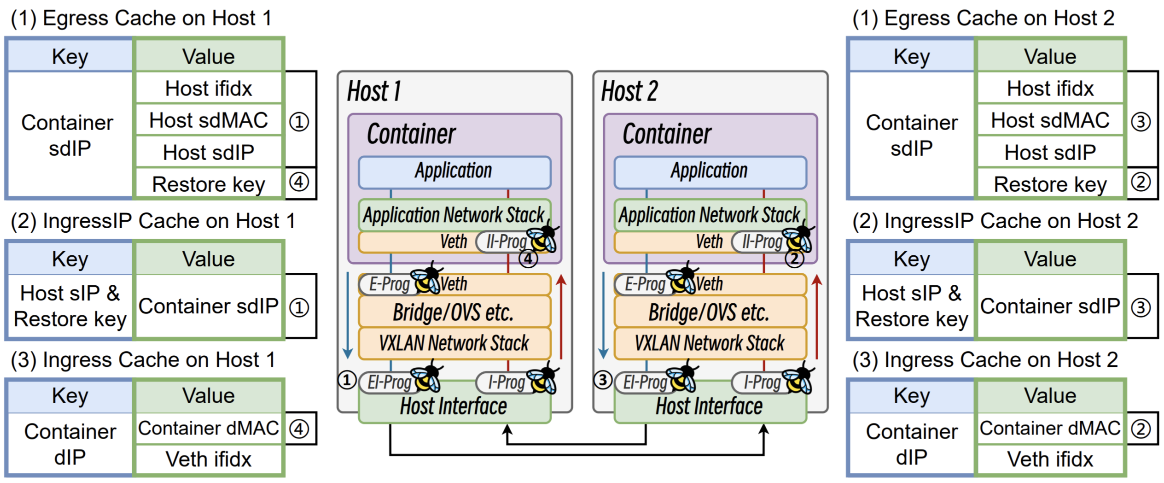


Figure 3.2: The cache initialisation of ONCache-t [19].

Figure 3.2 shows which interfaces these programs are attached to in the standard VXLAN overlay network setup.

To understand how ONCache works, we will walk through how packets flow between host 1 and host 2 and how caching certain invariants allows ONCache to eliminate the overhead.

Suppose the container on host 1 wants to send a packet to the container on host 2. The containers network stack will first build the packet and send it via its virtual ethernet device. This will first be processed by the egress fast path program on the virtual ethernet peer device, which resides in the host network namespace. The egress fast path program checks the egress cache, which at this point is empty. The packet is marked with a cache miss flag and allowed to proceed as normal.

When the packet reaches the host network interface, it will be processed by the egress cache init program. The egress cache init program will check the packet for the cache miss flag. If present, it will initialize the egress cache and the ingressIP cache. The idea here is to inspect both the outer and inner IP headers of the VXLAN packet and cache the mapping between container IP addresses and host IP addresses. This allows future invocations of the egress fast path program to rewrite the inner packet IP header and redirect it to the host interface, bypassing VXLAN encapsulation and processing by the bridge and VXLAN network stack. The egress cache init program also generates a restore key which it expects to receive in future ingress packets so that it can restore the IP header to the original container IP address.

When host 2 receives the packet, the ingress fast path program is executed. This program needs a mechanism to 'undo' the IP header rewriting done in the egress fast path program, so that the packet can be delivered to the correct container. This is the job of the ingress cache init program, which records the restore key generated by the egress cache init program on the other host. The egress fast path program then marks IP packets with this restore key so that the other host can reverse the IP header rewriting.

The authors are able to show that the throughput and latency of ONCache-t is comparable to that of bare-metal, and performs slightly better than Slim whilst also being compatible with all layer 4 protocols rather than just connection oriented ones.

3.2.2 Limitations

ONCache is designed to maintain compatibility and flexibility, meaning that there are very few limitations to its design. However, there is at least one important limitation in the ONCache-t design, which is that it requires using the DSCP field in the IP header to store the restore key.

This is an 8 byte field that in randomly initialized in the egress cache init program. This limits the number of concurrent connections that can be established between two containers whilst still benefiting from the performance improvements. This seems like an acceptable trade-off for most applications, but is worth noting.

3.3 Particle

Particle proposes an alternative approach for configuring virtual network devices and network namespaces in a way that allows for launching hundreds to thousands of containers in parallel substantially faster than is possible with standard VXLAN-based overlay networks.

3.3.1 Quantifying Network Namespace Setup Overhead

The authors of Particle use eBPF to pinpoint the exact bottleneck in creating and setting up network namespaces for containers. In particular, they focus on the overhead incurred when launching multiple containers in parallel on the same node.

Step	**Time (s)**	**Percent of Total**
Creating namespace	0.10	0.92%
Create VETHs	0.10	0.92%
Moving VETHs between namespaces	9.95	91.66%
Misc	0.71	6.48%

Table 3.2: Breakdown of the time spent by the kernel in setting up 100 network namespaces, adapted from [16]

They note that the majority of the time spent in setting up network namespaces is in moving virtual ethernet devices between namespaces. This is because doing so requires holding a global lock (rtnetlink) that prevents this operation from happening in parallel. This means all 100 containers must have their network namespaces setup sequentially.

3.3.2 Design and Implementation

To address the overhead identified, the authors propose that network namespaces are instead allocated on a per-tenant basis (rather than per-container), and that a single virtual ethernet device is used for all containers. Multiple IPs can be attached to the same virtual ethernet device to allow for different applications to be assigned different IPs.

The authors argue that dispensing with network namespace for each container is acceptable because in the context of burst-parallel workloads, the containers are working together to achieve a common goal under the management of the same application.

Particle results in a speed-up of 17x when creating 100 containers, and a speed-up of 213 with 1000 containers. Particle only needs to create a single namespace and move on virtual ethernet device from the host namespace to the tenant namespace, which allows for significant parallelism in the creation of containers.

3.3.3 Limitations

Despite the significant performance improvements, Particle has some limitations that prevent it from overcoming all the overheads of network namespaces.

- **Multi-tenancy**: Particle does not provide any benefit in the case where a cloud provider is hosting hundreds or thousands of tenants on a single machine - in this case each tenant would still need its own network namespace.

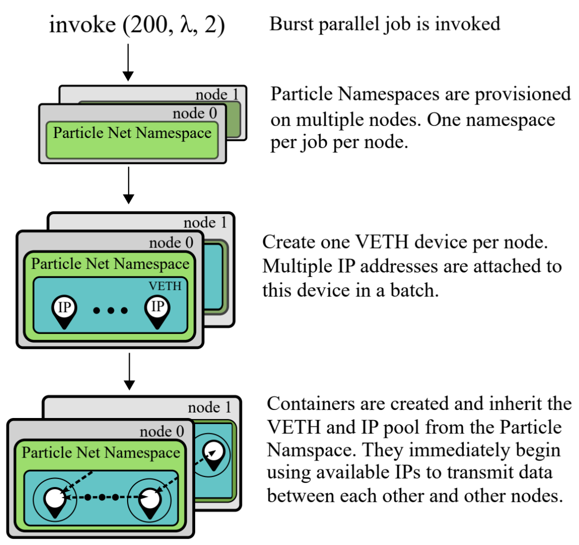

invoke (200, λ, 2)
Burst parallel job is invoked
node 1
node 0
Particle Net Namespace
Particle Namespaces are provisioned on multiple nodes. One namespace per job per node.
node 1
node 0
Particle Net Namespace
VETH
IP
IP
Create one VETH device per node. Multiple IP addresses are attached to this device in a batch.
node 1
node 0
Particle Net Namespace
Containers are created and inherit the VETH and IP pool from the Particle Namspace. They immediately begin using available IPs to transmit data between each other and other nodes.

Figure 3.3: Particle namespace with containers attached [16].

- **Latency Critical Applications**: The time is takes to set up a single container is unchanged, meaning that Particle does not provide any benefit for single container latency critical applications
- **Communication Overhead**: Particle does not address (or attempt to address) the communication overheads of overlay networks

3.4 Summary

Slim, ONCache and Particle all address overheads associated with the use of network namespaces in overlay networks, but all fundamentally still rely on the usage of network namespaces and aim to subvert the overheads rather than eliminating or reimagining their cause altogether.

This motivates the need for a new approach to container network isolation that no longer relies on network namespaces and virtual network devices.

Chapter 4

eBPF System Call Interception

This chapter presents the design and implementation of a set of eBPF hooks that allow for intercepting system calls and rewriting their arguments. These hooks are attached at the cgroup level, meaning they can easily be attached to containers. In section 4.1, we describe the user-facing interface of the hooks exposed to eBPF developers and the actions that the eBPF programs can perform. Then, in section 4.2, we describe the kernel modifications required to enable safely rewriting of kernel space copies of system call arguments.

4.1 Design

Existing eBPF hooks for system calls [22] are limited to a very small subset of socket related operations, and are further limited to the `AF_INET` address family. We introduce a new set of eBPF hooks that enable attaching programs to the entry and exit points of any system call. The primary purpose of these hooks is to provide a kernel-space mechanism for rewriting system call arguments and return values. However, these hooks also support fine-grained filtering of system calls based on deep argument inspection.

4.1.1 Interface

We define a new eBPF program type, `BPF_PROG_TYPE_CGROUP_SYSCALL`, along with a family of attach types that fall under this program type. For each system call, there are two associated attach types: one for the entry point and one for the exit point. These are named `BPF_CGROUP_SYSCALL_<syscall>` and `BPF_CGROUP_SYSCALL_<syscall>_EXIT`, respectively. This follows the pattern established by `BPF_PROG_TYPE_CGROUP_SOCK_ADDR` [22], which is an existing eBPF hook that allows for rewriting a small subset of networking related system call arguments. These hooks are attached to cgroups rather than to the system as a whole. This allows them to be selectively applied to a collection of processes — such as those belonging to a container — without requiring the eBPF program to implement its own process tracking logic. We allow multiple eBPF programs to be attached to the same hook in the same cgroup, and run the programs in the order that they were attached.

4.1.2 Program Context

The context passed to each eBPF program depends on the particular system call being intercepted (defined by the attach type). We design the context structure to closely mirror the original system call arguments, while introducing an additional field representing the return value. For entry point hooks, this field can be set by the eBPF program to override the system call return value and short-circuit execution. For exit point hooks, it reflects the actual return value and can also be modified before being returned to userspace.

The BPF_PROG_TYPE_CGROUP_SOCK_ADDR hooks use a single shared context structure for all attach types. This is reasonable because the attach types all relate to AF_INET socket operations and share similar arguments. In contrast, we are proposing a more generic set of hooks that can be used to intercept any system call, making it more appropriate to define a distinct context structure for each system call. This approach enforces a clear separation between attach types and provides eBPF developers with an explicit, unambiguous view of the accessible system call arguments.

```
struct bpf_cgroup_syscall_socket {
    __u32 family;
    __u32 type;
    __u32 protocol;
    __s32 ret;
};
```

Listing 4.1: The context provided to an eBPF program attached to the socket system call hook.

4.1.3 Kernel Hooking Points

It is essential that entry point hooks run as early as possible in the system call handler, and exit point hooks run immediately before the system call returns to user-space. This ordering allows eBPF programs to observe and modify system call arguments prior to any kernel validation checks, which means the modifications made by the eBPF programs undergo the same validation as arguments passed to the kernel by user-space applications. To protect against time-of-check-to-time-of-use race conditions (discussed in subsection 4.2.2), these hooks must not provide any pointers to user-space memory in the context structure. Instead, the system call handler should copy the user-space memory into kernel space and then pass a kernel pointer to the eBPF program.

4.1.4 Program Behaviour

The eBPF programs have read-write access to all the system call arguments via the context structure. Provided the program passes the verifier, there are no restrictions on how it manipulates these arguments. To short-circuit the system call, the program must set the return value field in the context and return BPF_SYSCALL_EXIT. Otherwise, returning BPF_SYSCALL_PASS will allow the system call to proceed as normal, potentially with modified arguments.

```
SEC("cgroup/syscall_socket")
int bpf_ns_socket(struct bpf_cgroup_syscall_socket *ctx)
{
    switch (ctx->family) {
    case AF_INET6:
        /* Allow the creation of AF_INET6 sockets */
        return BPF_SYSCALL_PASS;
    case AF_INET:
        /* Rewrite the socket family to AF_INET6 */
        ctx->family = AF_INET6;
        return BPF_SYSCALL_PASS;
    default:
        /* Block the creation of all other sockets with a permission denied
         * error. */
        ctx->ret = -EPERM;
        return BPF_SYSCALL_EXIT;
    }
}
```

Listing 4.2: An example of the `socket()` system call entry point hook being used to short-circuit with a permission error for all sockets except `AF_INET` and `AF_INET6`. All `AF_INET` sockets have their family argument changed so that the kernel instead creates a `AF_INET6` socket.

4.2 Implementation

To support the proposed eBPF hooks for rewriting system call arguments, we extend the Linux kernel with a new eBPF program type, as outlined in subsection 4.1.1. Once the program type is implemented, support for individual system calls can be added through a straightforward, reusable pattern. Accordingly, we separate the implementation of the program type (subsection 4.2.1) from that of the attach types (subsection 4.2.5). This section also covers key considerations, including TOCTTOU mitigations in subsection 4.2.2, context safety in subsection 4.2.3, and symbolic context access resolution in subsection 4.2.4.

4.2.1 Program Type Definition

Before adding any system call specific hooks, we first need to introduce a new eBPF program type, `BPF_PROG_TYPE_CGROUP_SYSCALL`. New eBPF program types are defined in the header file `include/linux/bpf_types.h` using the `BPF_PROG_TYPE` macro. We place our new program type alongside other cgroup related program types, which are wrapped in an `#ifdef` so that they are only compiled if the kernel is configured with the `CONFIG_CGROUP_BPF` option, as shown in Listing 4.3.

```
#ifdef CONFIG_CGROUP_BPF
...
BPF_PROG_TYPE(BPF_PROG_TYPE_CGROUP_SYSCALL, cg_syscall, __u64, u64)
#endif
```

Listing 4.3: The new eBPF program type definition. The first argument defines the type name, the second argument is a prefix used to identify program specific verification functions, and the last two arguments define the program return type.

We must also explicitly add the new program type in the `include/uapi/linux/bpf.h` header file, so that it is visible to userspace programs. This is done by updating the `enum bpf_prog_type` with the `BPF_PROG_TYPE_CGROUP_SYSCALL` type. There are a number of switch statements in the kernel where we need to add support for the new program type. In the `bpf()` system call handler, we update a function that defines which attach types are supported for our new program type. For now, we don't support any attach types. We also update a function which defines how the program should be attached and detached, which is done by calling an existing function that handles cgroup program types. Finally, we must update the verifier to place restrictions on the return value of our new program type. We restrict the return value to be either `BPF_SYSCALL_PASS` or `BPF_SYSCALL_EXIT`.

The BPF verifier relies on a set of helper functions that must be implemented for each new program types. These must be prefixed with the prefix specified in the program type definition, which in our case is `cg_syscall` (see Listing 4.3). The first helper function we defined is `cg_syscall_func_proto()` which defines the eBPF helper functions that a program type can access. We allow our programs to access all generic eBPF helper functions (such as accessing the PID of the calling process), but don't give them access to any helper functions that are typically more restricted. We discuss the implementation of `cg_syscall_is_valid_access()` and `cg_syscall_convert_ctx_access()` in subsection 4.2.3 and subsection 4.2.4 respectively.

We also define a macro, as shown in Listing 4.4, which will be used to cleanly insert eBPF hooks into the system call code. This macro takes a `type_macro` argument which is used to identify the desired attach type and a `fn_suffix` argument which is used to identify the function responsible for constructing the eBPF program context struct (specific to the particular attach type). The `__VA_ARGS__` argument is used to pass system call arguments.

```
#define __BPF_CGROUP_RUN_PROG_SYSCALL(type_macro, fn_suffix, ...)   \
({                                  \
    u32 __flags = 0;                        \
    int __ret = 0;                          \
    int ret_val = 0;                        \
    if (cgroup_bpf_enabled(CGROUP_SYSCALL_##type_macro)) {  \
        __ret = __cgroup_bpf_run_filter_syscall_##fn_suffix(__VA_ARGS__,
    ↪ &ret_val, &__flags); \
        if (__flags & BPF_SYSCALL_EXIT) {       \
            return ret_val;             \
        }                           \
    }                               \
    __ret;                              \
})
```

Listing 4.4: A macro which will be used to invoke our eBPF program at specific hooking points.

4.2.2 Context Safety and TOCTTOU Mitigations

Some system calls take pointers to user space memory as arguments. For example, the `bind()` system call takes a pointer to a `struct sockaddr` which defines the network interface that the socket should be bound to. Our eBPF hooks need to be able to read and modify these arguments before the system call is executed. Directly modifying user space memory is not safe, because a malicious program could attempt to modify the user space memory after our eBPF program does so, but before the system call is executed. This is known as a time-of-check to time-of-use (TOCTTOU) race condition. To protect against this, we carefully place our eBPF hooks so that we can provide our eBPF programs with a kernel space copy of the user space memory. Fortunately, the kernel already does this copying into kernel space memory. Listing 4.5 shows an example of how we place the eBPF hook for the `bind()` system call. The hook is placed immediately after the kernel has copied the user space memory into kernel space memory, but before the kernel checks that the file descriptor is valid and that the socket is of the correct type.

```
int __sys_bind(int fd, struct sockaddr __user *umyaddr, int addrlen)
{
    struct socket *sock;
    struct sockaddr_storage address;
    CLASS(fd, f)(fd);
    int err;

    err = move_addr_to_kernel(umyaddr, addrlen, &address);
    if (unlikely(err))
        return err;

    BPF_CGROUP_RUN_PROG_SYSCALL_BIND(&fd, &address, &addrlen);

    if (fd_empty(f))
        return -EBADF;
    sock = sock_from_file(fd_file(f));
    if (unlikely(!sock))
        return -ENOTSOCK;

    return __sys_bind_socket(sock, &address, addrlen);
}
```

Listing 4.5: The `bind()` system call hook is executed with a `struct sockaddr_storage` which resides in kernel space to protect against TOCTTOU race conditions.

4.2.3 Verifier Enforcement of Context Accesses

The eBPF verifier is responsible for ensuring that eBPF programs are only allowed to make verified accesses to the context structure. For example, they do not allow arbitrary reads of any pointers located within the context structure. For example, if you want to read or write to a system call argument, you are allowed read and write access a pointer that points to the beginning of that argument, but not necessarily halfway through. For our attach types, we must define which accesses are valid with the `is_valid_access()` function.

We want the verifier to allow most reads and writes to the context structure. Our hooks allow for the rewriting of system call arguments, which mean the programs should have flexible access to those arguments. This means that the function should return true whenever a read or write access is made to any offset corresponding to the field representing a system call argument. Furthermore, if the access is to an array, we should allow read and write access to any byte within that array.

4.2.4 Symbolic Context Access Resolution

To allow eBPF programs to read and write to the provided context struct, the kernel developer must implement a function called `convert_ctx_access()`. The verifier calls this function to replace symbolic references to the exposed context struct fields with the appropriate eBPF bytecode instructions that can be used to read and write to kernel memory. This function is provided with its own internal context struct that contains pointers to the kernel memory corresponding to the fields in the user-facing context struct exposed to eBPF programs. Listing 4.6 shows this internal context struct for the `bind()` system call hook.

```
struct bpf_cg_syscall_bind_kern {
    u32 *fd;
    struct sockaddr_storage *addr;
    u32 *addrlen;
    s32 *ret;
    /* Temporary "register" to make indirect stores to fields defined above.
     * We need three registers to make such a store, but only two (src and dst)
     * are available at convert_ctx_access time
     */
     u64 tmp_reg;
};
```

Listing 4.6: The kernel keeps an internal context struct that stores pointers to the kernel memory containing the system call arguments and return value

Whenever the verifier encounters a read that references a field inside the context struct, it calls the `convert_ctx_access()` function with the offset within the context struct of the field being accessed and a destination register where the value should be loaded into. The convert_ctx_access() uses this offset to find the corresponding field in the internal context struct and then emits two BPF_LOAD instructions: the first one loads the kernel memory address into the destination register and the second one dereferences that address and loads the value into the destination register.

Writing to a field in the context struct is slightly more complicated. In this case, we also have a source register containing the value that we want to write to kernel memory. The destination register contains the offset of the field within the context struct. Since the write should not have any side effects on any registers, we need to use another BPF register to store the pointer to the kernel memory that we want to write to. This is done by picking a BPF register (that isn't either the source or destination register) and storing its value in a temporary field in the internal context struct. We then load the kernel pointer into this register and write the value from the source register. Finally, we restore the value of the temporary register so that the write does not result in any side effects.

One limitation is that the convert_ctx_access() function is not able to support writing variable sized values. For example, we couldn't allow for both 4 byte and 16 byte writes to the struct sockaddr field. This makes dealing with polymorphic types (such as struct sockaddr) in the context struct difficult. To work around this, we only allow single byte read and writes to the struct sockaddr field. This allows maximum flexibility, but requires the eBPF programs to implement their own helper functions to interpret the byte array as a struct sockaddr.

4.2.5 Attach Point Integration

Now that the program type has been defined, we can start adding support for specific system calls by defining attach types. This section walks through the implementation of the connect() system call hook to demonstrate the reusable pattern used to add support for new system calls. The process is as follows:

1. Define the new attach type, the internal context structure and the eBPF program facing context structure
2. Implement the function that prepares the internal context structure
3. Update the context access verification function and the symbolic context access resolution function
4. Place the hook in the appropriate location inside the system call handler

For the connect() system call, we first define the attach type BPF_CGROUP_SYSCALL_CONNECT in the include/linux/bpf-cgroup-defs.h and include/uapi/linux/bpf.h header files. Next, we defined the internal context structure and the eBPF facing context structure:

```
struct bpf_cg_syscall_connect_kern {
    u32 *fd;
    struct sockaddr_storage *addr;
    u32 *addrlen;
    s32 *ret;
     u64 tmp_reg;
};

struct bpf_cg_syscall_connect {
    __u32 fd;
    unsigned char ss_data[128];
    __u32 addrlen;
    __s32 ret;
};
```

Inside the `kernel/bpf/cgroup.c` file, we implement the function that is called by the hook macro (see Listing 4.4) to prepare the internal context structure and invoke the eBPF program. The internal context structure is passed to the verifier so that it can be used to resolve symbolic references to the eBPF facing context structure at verification time. We use a read lock to ensure safe access to the cgroup of the current process. Listing 4.7 shows the implementation of this function for the `connect()` system call hook.

```
int __cgroup_bpf_run_filter_syscall_connect(int *fd,
                    struct sockaddr_storage *addr, int *addrlen,
                    int *ret_val, u32 *ret_flags) {
    struct bpf_cg_syscall_connect_kern ctx = {
        .fd = fd,
        .addr = addr,
        .addrlen = addrlen,
        .ret = ret_val,
    };
    int ret;

    rcu_read_lock();
    struct cgroup *cgrp = task_dfl_cgroup(current);
    ret = bpf_prog_run_array_cg(&cgrp->bpf, CGROUP_SYSCALL_CONNECT, &ctx,
        bpf_prog_run, 0, ret_flags);
    rcu_read_unlock();

    return ret;
}
```

Listing 4.7: Delegation function that prepares the internal context structure and runs all eBPF programs attached to the connect system call hook.

To update the context access verification function, we add a switch statement case for the `BPF_CGROUP_SYSCALL_CONNECT` attach type and allow access to the offsets of the start of each field in the eBPF facing context structure. The symbolic context access resolution function is also updated to support a switch statement case for the attach type. We define two macros that can be used to provide read and write access to the fields in the context struct, as shown in Listing 4.8. Finally we place a hook in the `__sys_connect()` function, which is the system call handler for `connect()`, as shown in Listing 4.9.

```
case BPF_CGROUP_SYSCALL_CONNECT:
    switch (si->off) {
        CG_SYSCALL_FIELD_RW_ACCESS(connect, fd, fd);
        CG_SYSCALL_FIELD_RW_ACCESS_RANGE(connect, ss_data, 127);
        CG_SYSCALL_FIELD_RW_ACCESS(connect, addrlen, addrlen);
        CG_SYSCALL_FIELD_RW_ACCESS(connect, ret, ret);
    }
```

Listing 4.8: Switch statement case for the `convert_ctx_access()` function for the `connect()` system call hook.

```
int __sys_connect(int fd, struct sockaddr __user *uservaddr, int addrlen)
{
    struct sockaddr_storage address;
    ...
    __BPF_CGROUP_RUN_PROG_SYSCALL(CONNECT, connect, &fd, &address, &addrlen)
    ...
    return __sys_connect_file(fd_file(f), &address, addrlen, 0);
}
```

Listing 4.9: We invoke the eBPF program for the connect attach type, passing pointers to the system call arguments.

Chapter 5

Lightweight Alternative to Network Namespaces

This chapter describes the design and implementation of a lightweight alternative to network namespaces that relies on the eBPF system call argument rewriting hooks introduced in chapter 4. We show how these hooks can be used to provide network isolation for containers whilst allowing them to remain in the host network namespace. This design removes the need to move virtual ethernet devices between network namespaces, which the authors of Particle [16] found to be the most significant bottleneck in container launch time. Furthermore, we show how our hooks can be used to create an eBPF fast-path for pod-to-pod overlay style communication that achieves throughputs and latencies comparable to processes communicating directly through the main host network interface.

5.1 Design

We propose a design that places all containers in the host network namespace and use our eBPF system call hooks to control which network interfaces each container can see and use. By intercepting and rewriting socket system call arguments, we can force containers to use a specific virtual ethernet device and block access to other network interfaces. Furthermore, we can control the view of the network state that each container sees by using our hooks to intercept netlink messages and filesystem accesses. We also propose a communication fast-path that allows containers restricted access to the host network interface, potentially substantially improving the throughput and latency of pod-to-pod communication.

5.1.1 Overview

Network namespaces can be considered to be an "ownership" based approach to network isolation because they place the restriction that network interfaces must belong to only one network namespace at a time. Our design is closer to a "policy" based approach because all processes reside in the host network namespace, and we use our eBPF system call hooks to control which network interfaces each process can see and use. To ensure that the containers only use the virtual ethernet devices assigned to them (i.e `red-in` and `blue-in` in Figure 5.1), we intercept and rewrite the arguments to the `bind()` and `connect()` system calls. To prevent the containers from seeing or modifying the host network state (including host network interfaces and routing rules), we intercept the creation of all netlink sockets with the `socket()` system call hook and redirect all communication to an user-space netlink server. We hide network state revealed by the file system in locations such as `/sys/class/net` by intercepting system calls related to filesystem access, such as `open()` and `read()`. We also take advantage of the containers residing in the host network namespace by using our eBPF system call hooks to allow the containers to safely access the host network interface directly. This avoids the overhead of the VXLAN overlay.

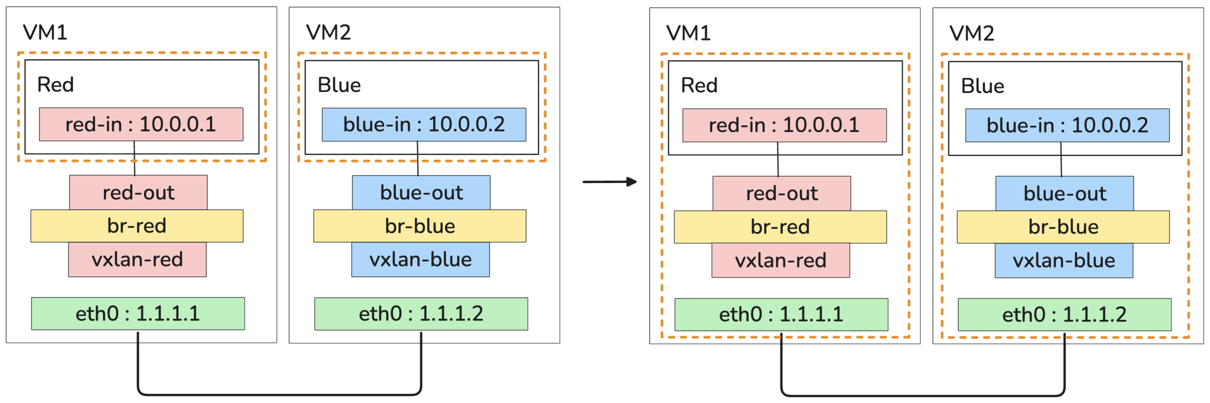


Figure 5.1: Red and blue are containers, residing on VM1 and VM2 respectively. The dotted orange box shows the network namespace of the container. On the left, the container has its own separate network namespace. On the right, the container resides in the host network namespace which means it is able to see, use and modify all network interfaces.

5.1.2 Restricted Network Access

The containers in Figure 5.1 should only be able to use the network interfaces that have been allocated to them (i.e. `red-in` and `blue-in`). They should not be able to bind a socket to `eth0`, or any other network interface in the host network namespace. We enforce this with the following two eBPF system call hooks:

- `bind()`: if a container tries to bind to the allocated network interface (e.g. `red-in`), we allow the bind system call to proceed as normal. If it tries to bind to `INADDR_ANY`, we rewrite the IP address to the IP address allocated to the container. Any other IP address results in an `EADDRNOTAVAIL` error being returned.

- `connect()`: we use the `bpf_bind()` helper function to force the client socket to first bind to the allocated network interface, which prevents the kernel from falling back to the main host network interface whenever a container attempts to connect to a remote host outside the overlay network.

These two hooks allow us to provide containers with the ability to use TCP sockets to communicate over the overlay network. This design can be extended to support other protocols by intercepting system calls such as `sendto()` and `recvmsg()`. A further consideration is how to allow multiple containers to use the same IP address. Traditionally, each container has a virtual ethernet device inside its network namespace. This virtual ethernet device can be assigned any IP address, unless there is already a device with that IP address in the same network namespace. Since we are placing all container network interfaces in the host network namespace, we need to deal with the case where two containers request the same IP address.

Our eBPF hooks provide the flexibility to handle this. In an eBPF map, we can store an entry for each container that contains the IP address that the container wants to use, as well as a potentially different underlying IP address which is the one actually assigned to the virtual ethernet device. This requires a slight modification to the existing `bind()` and `connect()` eBPF programs, as well as the introduction of hooks for `getsockname()` and `getpeername()`. We should intercept these two system calls to overwrite the IP address returned to the container so that it matches the IP address that the container requested, rather than the underlying IP address assigned to the virtual ethernet device.

5.1.3 Userspace Netlink Server

Netlink is the mechanism through which user-space processes can communicate with the kernel to query and modify the network state. We need to be able to intercept netlink messages so that we can filter out information about the host network state (e.g. to hide the host network interfaces from the container) and block certain netlink operations such as those to delete network interfaces or change IP routing rules.

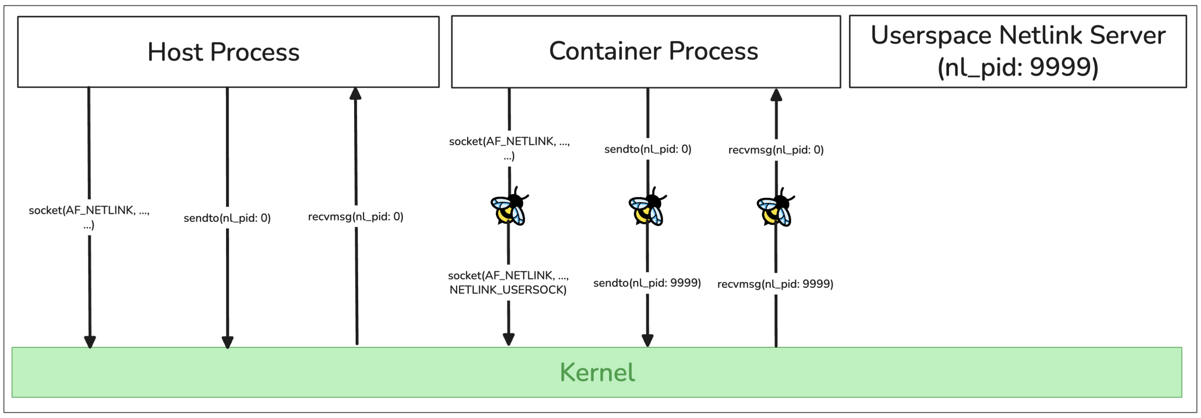


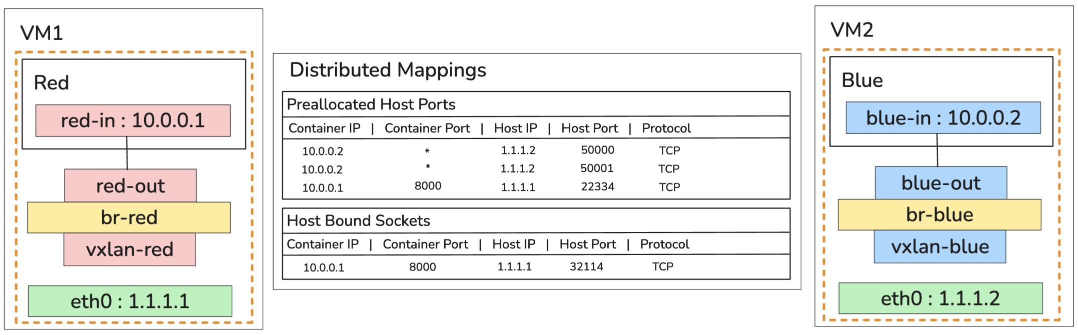
Figure 5.2: By intercepting the `socket()`, `sendto()` and `recvmsg()` system calls, we can redirect netlink communication to a userspace netlink server over the `NETLINK_USERSOCK` protocol, bypassing the kernel.

Using our generic eBPF system call hooks, we can intercept the `socket()` system call and overwrite the protocol of all netlink sockets to `NETLINK_USERSOCK`. This is a protocol that is reserved for the communication of two user-space applications (using the netlink API) rather than communication between the kernel and user-space. This is incredibly powerful because it means that we can redirect all netlink communication to a trusted user-space netlink server that runs on the host, and completely bypass the kernel to ensure that no information about the host network state could be leaked to the container. By also intercepting the `sendto()` and `recvmsg()` system calls, we can update the `nl_pid` field in the netlink header such that the message is sent to the user-space netlink server and such that the application is tricked into thinking the message came from the kernel, rather than another user-space process (`nl_pid` of zero represents the kernel).

Figure 5.2 provides an overview of how this works: processes inside the container with the eBPF hooks attached will have their netlink communication redirected to the user-space netlink server, whereas non-container processes will continue to communicate directly to the kernel. Since the user-space netlink server runs on the host, it has the ability to send netlink messages to the kernel and therefore perform actions on behalf of the container (if deemed appropriate). The user-space netlink server can also entirely spoof responses to container processes. As an example, this allows us to prevent commands like `ip link` from exposing the host network interfaces, and instead we can return a fake list of network interfaces that gives the container an isolated view of the network state.

5.1.4 Communication Fast Path

Our eBPF hooks allow for containers to completely avoid the overhead of communicating over the VXLAN overlay network by allowing them restricted access to the host network interface. When a container tries to bind a socket to a network interface, we first check that the requested IP address matches the one that it has been allocated. Then, we rewrite the arguments to bind the socket directly to the host network interface (i.e. eth0 in Figure 5.3), which includes assigning any free port (not necessarily the one that the container requested). We distribute this mapping to all nodes in the cluster using a distributed key-value store such as etcd (as shown in Figure 5.3). When a client attempts to connect to this socket, we can rewrite the arguments of the connect system call using this distributed mapping.



Preallocated Host Ports

Container IP	Container Port	Host IP	Host Port	Protocol
10.0.0.2	*	1.1.1.2	50000	TCP
10.0.0.2	*	1.1.1.2	50001	TCP
10.0.0.1	8000	1.1.1.1	22334	TCP

Host Bound Sockets

Container IP	Container Port	Host IP	Host Port	Protocol
10.0.0.1	8000	1.1.1.1	32114	TCP

Figure 5.3: We can allow the containers to bind to and connect from the host interfaces to allow a fast path. This requires distributing mappings for bound sockets, as well as a preallocated set of hosts ports for each container to connect from.

Server containers need a way of identifying which client container is attempting to connect to them. By pre-allocating a range of host ports to each container, the server container can determine which client container is making the connection request by looking at the source port of the incoming connection. This allows eBPF programs to write the correct container IP address in system call hooks such as `accept()` and `getpeername()`. One limitation with this approach is that it limits the number of concurrent connections that can be made to a single server socket from a single client container. This can be overcome by supporting falling back to the standard VXLAN overlay network when all preallocated ports are in use.

5.1.5 Filesystem Network State

Information about the network state is accessible through various locations in the filesystem. For example, the `/sys/class/net` directory contains a subdirectory for each network interface on the host. This subdirectory contains information about the network interface, such as its name, MAC address, and IP address. To provide robust network isolation to containers, we must ensure that they are unable to arbitrarily access this information. Our eBPF system call hooks offer a powerful mechanism to enforce this: we can intercept filesystem system calls such as `open()`, `read()`, and `readdir()` and either block them entirely or modify the data returned to the container so that they can only see the network interfaces that they are allowed to use.

5.1.6 Defence in Depth

Our eBPF system call hooks provide a mechanism for system call filtering based on deep inspection of system call arguments. To provide further protection against malicious containers, we can take advantage of the BPF LSM (Linux Security Module) hooks to enforce additional security policies. For example, a policy can be enforced that prevents containers from creating sockets with a particular family or protocol, or that prevents them from binding to certain IP addresses. By combining our eBPF system call hooks with BPF LSM, we can create an extra layer of security that ensures that rewriting of the system call arguments by the eBPF programs does not violate any isolation policies.

5.2 Implementation

We implement our lightweight alternative to network namespaces in a Kubernetes cluster running on top of a custom Linux kernel that has support for our eBPF system call hooks. We use a Kubernetes DaemonSet that runs a `go` application that we call the BPF agent. This `go` application uses the `ebpf-go` library [39] to load and attach our eBPF programs to the cgroup of each pod in response to Kubernetes API events. In this section, we first describe the architecture of the BPF agent and how it is used to manage network isolation for pods. Then, we give a detailed overview of the implementation of our eBPF programs. Finally, we briefly outline the implementation of our user-space netlink server and the communication fast path.

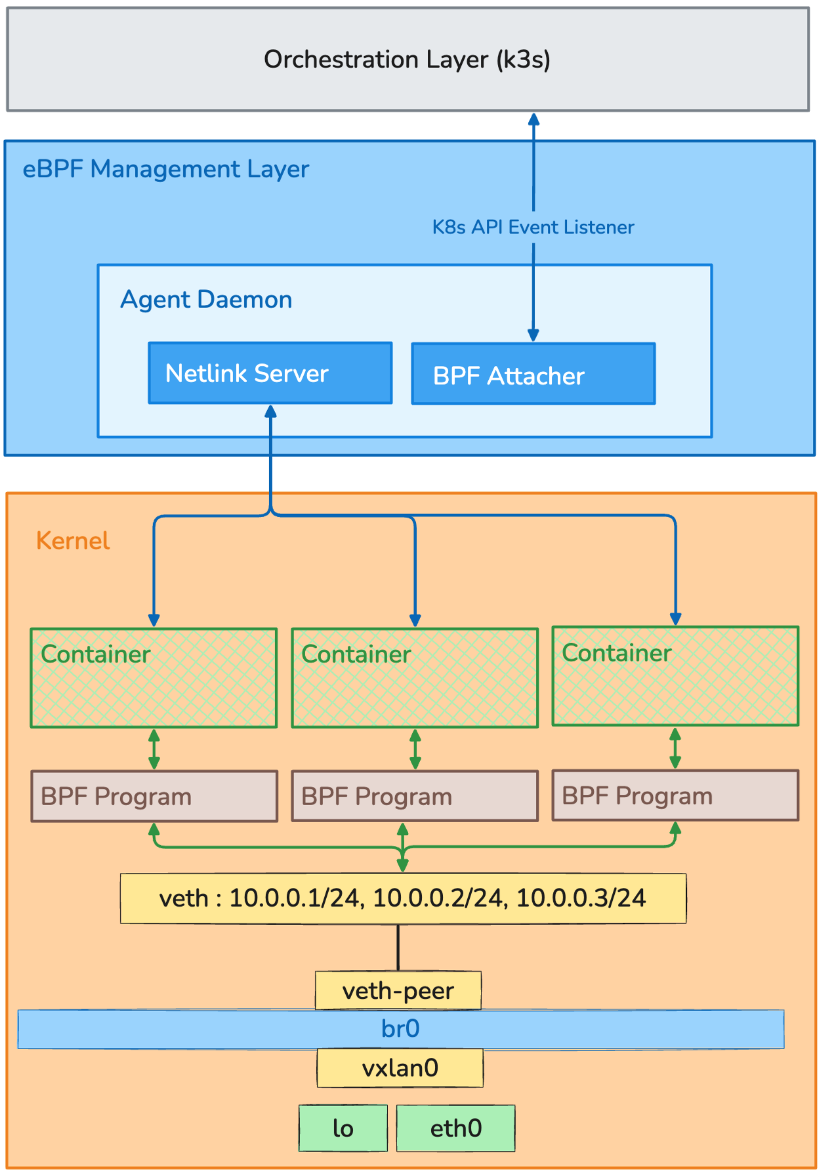


Figure 5.4: The Kubernetes API event listener sits in the eBPF management layer, and listens to events from the orchestration layer so that it can attach the eBPF programs to the pods whenever they are created.

5.2.1 Architecture

To understand how the BPF agent fits into our design, we separate the architecture into three layers: the orchestration layer (Kubernetes), the eBPF management layer and the kernel layer (as shown in Figure 5.4). The BPF agent sits in the eBPF management layer and has two main responsibilities. Firstly, it listens to Kubernetes API events and attaches eBPF programs to the cgroup of each pod as they are created. An IP address for the pod is also assigned to the shared virtual ethernet device. The second responsibility is to start a user-space netlink server that acts as a man-in-the-middle for all netlink requests made by processes running inside the pods.

The BPF agent is implemented as a Kubernetes DaemonSet, which means that it runs on every node in the cluster. The user-space netlink server is responsible for handling netlink requests from all pods running on the node, rather than having a separate netlink server for each pod. Each container has its own set of eBPF programs attached, rather than having a single set of eBPF programs that must disambiguate between different pods. Rather than loading and verifying the eBPF programs on each pod creation, the BPF agent loads and verifies the eBPF programs into the kernel when it starts up and attaches those programs to the cgroup of pods as they are created. The overhead of attaching eBPF programs to a pod's cgroup is minimal.

5.2.2 eBPF Programs

eBPF programs are attached to the cgroup of each Kubernetes pod to ensure that the pods have an isolated view of the network state. These hooks perform system cal argument rewriting to hide the underlying host network state from the pod and masquerade the pod's network interface as if it were in its own network namespace. Inspired by Particle [16], we use a single virtual ethernet device combined with a VXLAN interface to provide pod-to-pod communication. We use our eBPF programs to intercept networking related system calls and rewrite their arguments so that they bind to the correct virtual ethernet device using their allocated IP address. We implement the following system call hooks:

- `bind()`: if a process tries to bind a socket to the `INADDR_ANY` address, we rewrite the address to the underlying IP address on the virtual ethernet device associated with the pod. Since the pod IP address is not necessarily the same as the one assigned to the virtual ethernet device, we also perform this rewriting if the process attempts to bind specifically to its IP address.

- `connect()`: when a process tries to connect to a remote pod, we rewrite the destination address to match the IP address assigned to the virtual ethernet device associated with the remote pod. We also force the socket to bind to its virtual ethernet device so that it uses the correct IP address when sending packets and so that the kernel does not automatically fall back to using the host network interface (e.g. if the pod tries to connect to an IP address outside the cluster). We apply a similar rewriting to the `sendto()` and `recvmsg()` system calls.

- `getsockname()`: we rewrite the IP address and port numbers returned by this system call to match the IP address of the pod (rather than the IP address assigned to the virtual ethernet device). We also do this for the `getpeername()` and `accept()` system calls.

With these eBPF programs, we can provide a minimal set of networking functionality to the pods whilst giving them the illusion of network isolation. As mentioned in subsection 5.1.5, for complete network isolation we also require eBPF programs to intercept the `open()`, `openat()` and `read()` system calls to ensure that network state cannot be leaked through the file system.

The Kubernetes `hostNetwork:true` configuration also places the pod in the host UTS namespace which means, for example, that the pod shares the host's hostname and domain name. To give the pod its own hostname and domain name, we implement an eBPF hook for the `uname()` system call. This hook rewrites the `struct utsname` returned by the system call. Similarly, we can hook into the `gethostname()` and `sethostname()` system calls.

5.2.3 Userspace Netlink Server

We implement support for a small subset of netlink messages in a user-space netlink server that runs as part of the BPF agent. The user-space netlink server first creates a `NETLINK_USERSOCK` socket and binds it to `nl_pid 9999`. This is a special netlink protocol that allows for user-space processes to communicate with one another. The netlink server then uses the `recvmsg()` system call to continuously listen for incoming netlink requests from processes running inside the pods. The server supports two netlink message types: `RTM_GETLINK` and `RTM_GETADDR`. These message types are used to query the network interfaces and IP addresses associated with the pod. Using the `nl_pid` inside the netlink message header, the server can determine which pod the request is coming from and can access the pod metadata eBPF map to construct an appropriate response. This completely bypasses the kernel, which means that no host network information can be leaked to the pod through netlink. To redirect netlink requests sent by processes running inside the pods to the user-space netlink server, we implement eBPF programs using our system call hooks that:

- Overwrite the protocol of all netlink sockets created with the `socket()` system call to `NETLINK_USERSOCK` so that it can communicate with the user-space netlink server.
- Rewrite the `nl_pid` header in `sendto()` system calls from `0` to `9999` so that the messages are routed to the user-space netlink server.
- Rewrite the `nl_pid` header in `recvmsg()` system calls from `9999` to `0` so that user-space processes inside the pods think that they are communicating with the kernel rather than the user-space netlink server.

The design and implementation of the user-space netlink server is flexible enough to support additional netlink message types. Furthermore, it can selectively forward requests to the kernel if needed, for example if a pod wants to create and use its own virtual network interface. The user-space netlink server could use standard netlink sockets to create this virtual network interface and then update eBPF maps to allow the pod to use it.

5.2.4 Communication Fast Path

Rather than solely relying on a VXLAN based overlay network for pod-to-pod communication, we take advantage of the fact that our pods reside in the host network namespace and the fact that our eBPF programs can allow the pods safe and restricted access to the host network interface. We do this by pre-allocating a set of ephemeral ports from the host network interface to each pod. In an eBPF map, we store metadata about the placement of all pods in the cluster. This allows us to find the underlying host IP address of any pod in the cluster. We also store the ephemeral ports assigned to each pod in this map. We extend our eBPF programs to do the following:

- If a process tries to `bind()` a socket to a pre-specific port (e.g. 8000), we rewrite the `struct sockaddr` to bind the socket to a preallocated ephemeral port on the host network interface (e.g. 50000)
- When a process attempts to `connect()` to port 8000 on the server pod, we first bind the client socket to a preallocated ephemeral port on the client pod's host network interface (e.g. 60000). We then rewrite the `struct sockaddr` so that the destination address matches

the IP of the server pod's host network interface and rewrite the port to be 50000. The server knows which pod is connecting to it because it has the preallocated port mappings (e.g. it knows which pod has been assigned port 60000).

- Similarly to before, we rewrite the returned values to the `getsockname()`, `getpeername()` and `accept()` system calls.

The advantage of this approach is that it allows pods to achieve host level network performance and avoid the overheads of VXLAN overlay. Implementing such a design with our eBPF hooks is straightforward in comparison to a similar design proposed by Slim [18], which requires an `LD_PRELOAD` library as well as a kernel module and doesn't support connectionless protocols.

Chapter 6

Evaluation

We evaluate the performance of our lightweight alternative to network namespaces using a suite of microbenchmarks and two real-world applications (`nginx` [25] and `postgres` [26]). Our testbed consists of a two-node cluster, where each node runs as a QEMU virtual machine on the same physical host. Each node has 12 CPU cores, 16 GB of RAM and runs our custom Linux kernel. The nodes are connected via two TAP interfaces and a Linux bridge. We compare our system's performance against two baselines: a standard Linux VXLAN overlay network and processes running directly on the host machine. Additionally, for container startup time, we compare our system to Particle [16] and for throughput and latency, we compare to our system to ONCache [19], which is a current state-of-the-art overlay network implementation.

6.1 Microbenchmarks

Before evaluating the performance of our lightweight alternative to network namespaces in real-world applications, we first conduct a series of microbenchmarks to characterize the performance of our eBPF-based network isolation mechanism. These microbenchmarks measure system call interception overhead and container startup time, as well the throughput and latency within an overlay network configuration.

6.1.1 Interception Overhead

The primary advantage of our eBPF system call interception mechanism is that it does not require binary rewriting and is therefore suitable for use by cloud providers. However, since eBPF programs are executed directly in the kernel, the performance overhead is likely to be fairly low. To quantify this, we perform an experiment where we measure the time taken to execute a system call and compare it to a baseline and existing user-space interception mechanisms.

Our experiment involves creating a socket and binding it to the loopback interface. We measure the time taken to execute the `bind()` system call using `CLOCK_MONOTONIC`. We repeat this experiment 1000000 times and record the average time. Our aim is to understand the performance overhead introduced by invoking an eBPF program. Therefore, the eBPF program does not perform any operations and returns immediately. As a baseline, we use an unmodified Linux kernel (`v6.15-rc1`). To understand the performance in comparison to state-of-the-art system call interception mechanisms, we run the same experiment with zpoline [7] and lazypoline [6]. These are also configured to return immediately. We do not compare our performance to any existing kernel-based system call interception mechanisms because they do not support argument rewriting and solely focus on system call filtering, therefore being unsuitable for our use case.

Figure 6.1 shows the results of our experiments. The baseline, as expected, takes the least amount of time. Our eBPF-based mechanism is 1.21× slower than the baseline, zpoline is 1.19× slower and lazypoline is 1.86× slower. However, zpoline is not a completely exhaustive

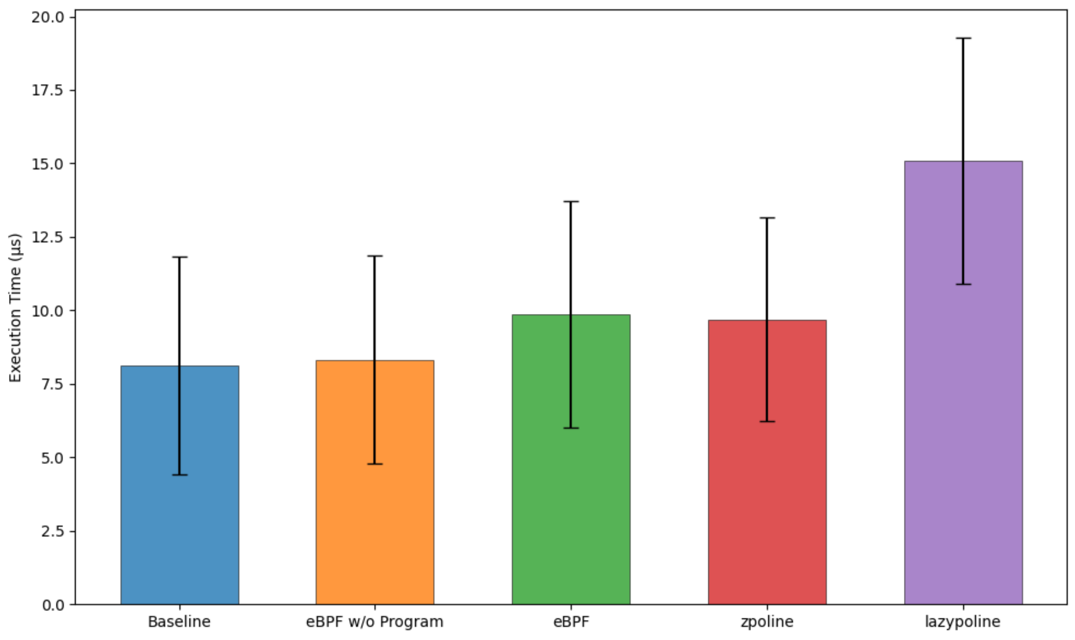


Figure 6.1: Average execution time of the `bind()` system call under various system call interception mechanisms.

interception mechanism and therefore cannot intercept all system calls [6]. In contrast, both lazypoline and our eBPF-based mechanism are completely exhaustive. These results are inline with the results found by lazypoline [6]. A significant limitation of our eBPF mechanism is that a small overhead is incurred for every system call made by all processes on the machine, regardless of whether the system call is being intercepted. This is because the kernel system call handler must check if any eBPF programs are hooked to the attachment point. User-space interception mechanisms do not incur this overhead, as the interception is performed at a per-process granularity. This is the `eBPF w/o Program` in Figure 6.1. To quantify the impact of this, we repeat our experiment without any eBPF programs attached. Our modified kernel is only 1.02× slower than the baseline, which suggests that the overhead of checking for eBPF programs is negligible compared to the overhead of executing the eBPF program itself. However, our experiment does not account for the time taken to execute the system call itself. This varies substantially depending on the particular system call, which means that the overhead could be more pronounced (relative to the system call execution time) for system calls that take less time to execute than `bind()`.

To summarize, our eBPF-based system call interception mechanism indeed incurs a small overhead compared to the baseline. However, the overhead is substantially lower than the current state-of-the-art fully exhaustive user-space interception mechanism. This can be attributed to the fact that our eBPF programs are executed in the kernel. Whilst zpoline has a slightly lower overhead, it is not exhaustive. Most crucially, our eBPF-based mechanism does not require any binary rewriting which makes it suitable for use by cloud providers.

6.1.2 Container Startup Time

We evaluate the startup performance of our lightweight alternative to network namespaces by measuring the time required to create a fully interconnected network of containers. We compare this to an equivalent setup that uses traditional network namespaces, as well as to the Particle [16] design, which represents the current state-of-the-art. Our experiments vary both the number of tenants (M) and the number of containers per tenant (N), as shown in Figure 6.2.

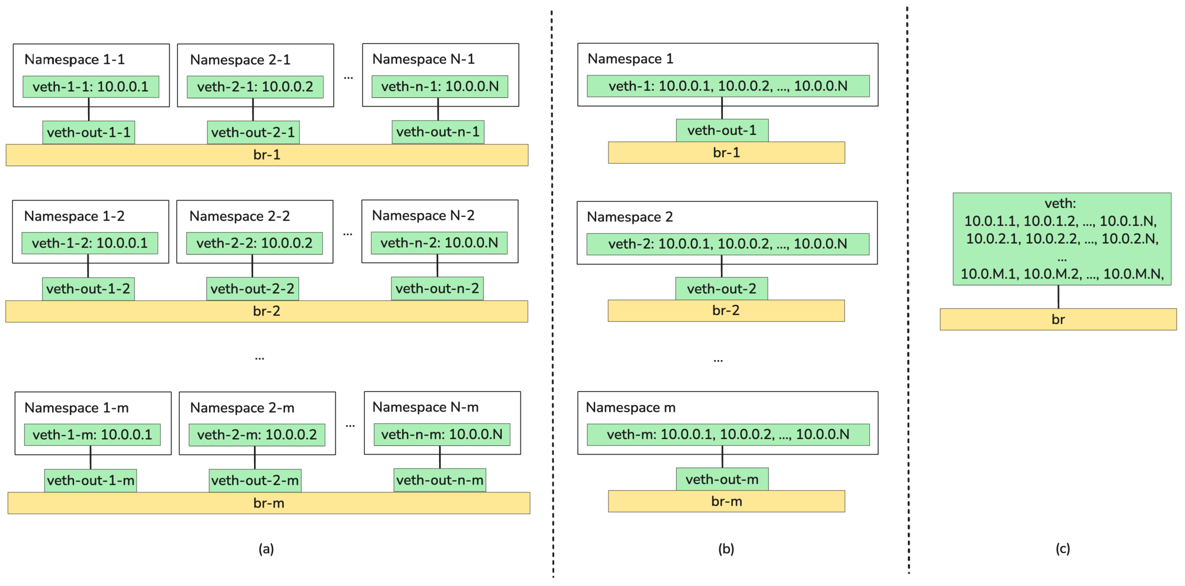


Figure 6.2: (a) Standard Linux Setup: each container is allocated its own network namespace and virtual ethernet device, (b) Particle: network namespaces and virtual ethernet devices are consolidated per tenant, with multiple IPs assigned to a single shared virtual ethernet device for each tenant, (c) by attaching eBPF programs to the cgroup associated with each container, we can create a single virtual ethernet device for all tenants and place all containers in the host network namespace

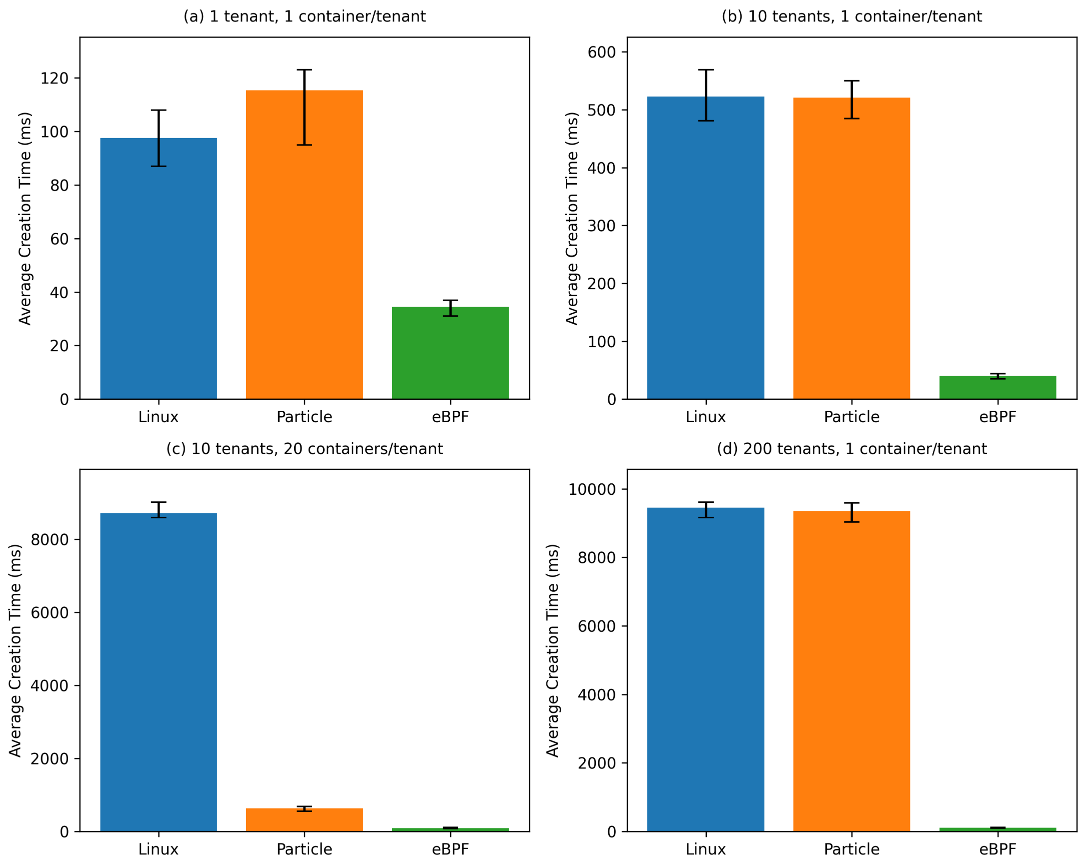


Figure 6.3: Summary of launch times for a selection of tenant and container count configurations.

The results, shown in Figure 6.3, demonstrate that our eBPF-based network isolation mechanism achieves consistently lower launch times compared to both the traditional Linux network namespaces and Particle. While a single container, single tenant setup shows a nearly 60% reduction in launch time with our eBPF-based mechanism, the benefits become even more pronounced as the number of containers and number of tenants increase. Particle is able to mitigate the overhead of creating a network namespace per container within a single tenant, but it still incurs a substantial overhead in the multi-tenant case. In contrast, our eBPF-based approach maintains consistently low launch times regardless of the number of tenants. For example, when launching one container across 200 tenants in parallel, our eBPF approach achieves a 92× reduction in launch time compared to Particle. These results are expected, because our eBPF-based approach does not require moving any virtual ethernet devices across network namespaces, whereas Particle still requires moving one virtual ethernet device per tenant.

6.1.3 Throughput and Latency

To understand the performance benefits of our lightweight alternative to network namespaces, we measure the throughput and latency between two containers residing on two different nodes and connected via an overlay network. We compare these measurements to those achieved by a standard Linux VXLAN overlay network. Additionally, we compare the throughput and latency of our communication fast path to that of two processes running directly on the hosts and to that of ONCache [19] which represents the current state-of-the-art in overlay networks.

To measure throughput, we use `iperf3` [40]. The client and server are pinned to a single CPU core using `taskset` [41]. The client uses a 1400B MSS, the cubic TCP algorithm and runs for 65 seconds (with the first 5 seconds being omitted from the results). We report the average, minimum and maximum results over ten repeats. To measure latency, we use `ping` [42] and report the average, minimum and maximum round-trip times (RTTs) over 100 repeats. The results for throughput are presented in Table 6.1 and for latency in Table 6.2. We also summarize the results in a bar chart in Figure 6.4.

	Throughput (Gbits/sec)				
	Linux VXLAN	**eBPF**	**Host**	**eBPF Fast-path**	**ONCache**
Min	2.20	2.21	28.40	28.40	27.20
Avg	2.30	2.33	29.58	29.17	27.83
Max	2.38	2.42	30.30	30.00	28.90

Table 6.1: Throughput measurements (in Gbits/sec) for different overlay network implementations. Our eBPF implementation is comparable to the standard Linux VXLAN, and our eBPF fast-path is comparable to host-to-host communication as well as ONCache.

	RTT (ms)				
	Linux VXLAN	**eBPF**	**Host**	**eBPF Fast-path**	**ONCache**
Min	1.064	1.018	0.972	0.782	0.891
Avg	1.277	1.348	1.092	1.121	1.161
Max	1.411	2.097	1.265	1.267	1.513

Table 6.2: Latency measurements (in ms) for different overlay network implementations. Our eBPF implementation achieves latency comparable to the standard Linux VXLAN, and our eBPF fast-path is comparable to host-to-host communication as well as ONCache.

The results suggest that the overhead of our eBPF system call interception programs is negligible: both the throughput and latency achieved by our (unoptimized) eBPF network isolation mechanism is comparable to that of a standard Linux VXLAN overlay network that uses network

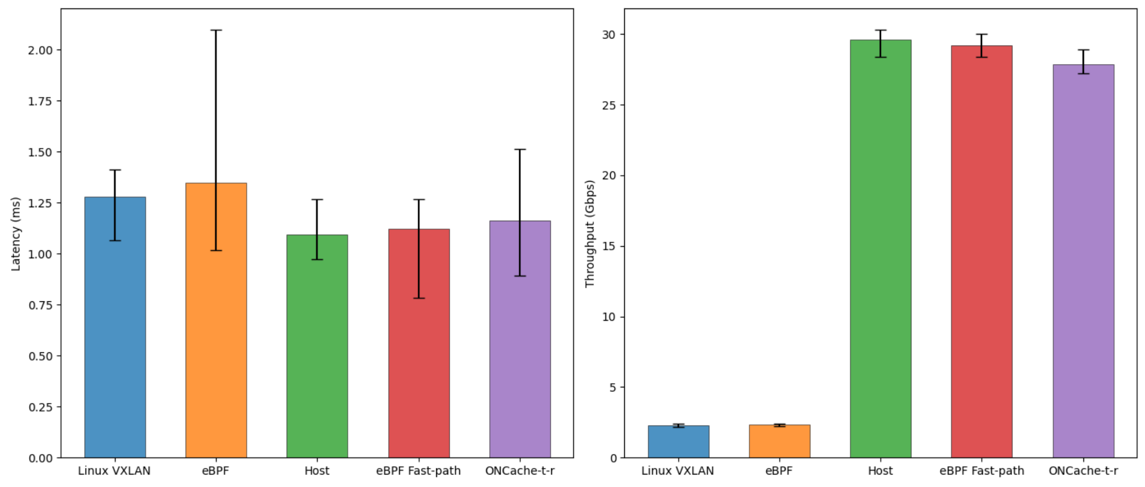


Figure 6.4: Summary of throughput and latency measurements for different overlay network implementations. Our eBPF implementation achieves throughput and latency comparable to the standard Linux VXLAN, suggesting that the overhead of our system call interception programs is negligible. Furthermore, our eBPF fast-path is able to achieve performance akin to that of processes communicating directly over the main host network interface, slightly outperforming the ONCache implementation.

namespaces. Furthermore, the eBPF fast-path achieves throughput and latency comparable to that of the host and slightly outperforms ONCache. Such an optimization is only possible due to the containers residing in the host network namespace, as the containers are able to have direct access to the host network interface. In contrast, ONCache places `tc` eBPF programs on the virtual ethernet devices which results in a per packet processing overhead, explaining the slightly lower throughput and higher latency compared to our eBPF fast-path.

A caveat to the results presented in Table 6.1 and Table 6.2 is that the Linux VXLAN measurement is significantly affected by the fact that our test bed involves two QEMU virtual machines [43]. In reality, the performance overhead of VXLAN has been found cause a 48% reduction in throughput and an 85% increase in latency [18].

6.2 Real-world Applications

To further evaluate our eBPF fast-path, we deploy two real-world applications: `nginx` [25] and `postgres` [26]. This gives us an understanding of the performance characteristics under two different real-world workloads, rather than just microbenchmarks.

6.2.1 nginx

We run an `nginx` [25] container on one node and the `wrk` [44] web server benchmarking tool on another container running on another node. The benchmarking tool is configured to spawn 12 threads and create 100 connections to the `nginx` server. The test runs for 60 seconds. The web server serves the default nginx welcome page, which is 612 bytes. We run this benchmarking tool for our eBPF fast-path and the standard Linux VXLAN overlay network, as well as directly on the host. The benchmarking tool gives us the latency and throughput (in requests per second) achieved. The results are presented in Table 6.3 and summarized in Figure 6.5.

These results are consistent with the results of our microbenchmarks: our eBPF fast-path is able to achieve throughput and latency comparable to that of the host, while the standard Linux VXLAN overlay network incurs a significant performance overhead. The reduction in latency and

	Host	**eBPF Fast-path**	**Linux VXLAN**
Latency (μs)	675.41 (+0.00%)	664.21 (-1.66%)	923.33 (+36.71%)
Requests/sec	161,606.12 (+0.00%)	162,368.76 (+0.47%)	106,149.57 (-34.32%)

Table 6.3: Comparison of throughput and latency achieved by `wrk` against an `nginx` web server.

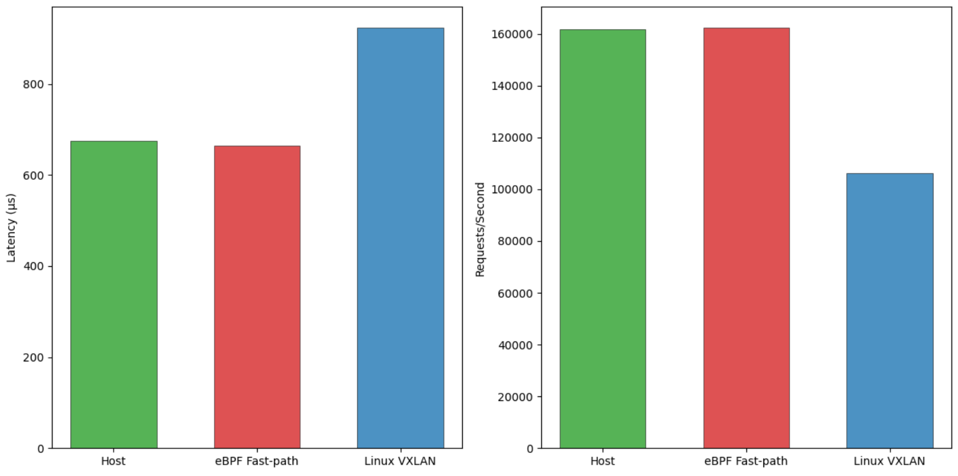


Figure 6.5: Comparison of throughput (right) and latency (left) achieved by `wrk`.

increase in throughput of our eBPF fast-path compared to the host is likely due to random noise and indicative of the negligible overhead of our eBPF system call interception programs. This is a substantial improvement over the standard Linux VXLAN overlay network, which incurs a 36.71% increase in latency and a 34.32% decrease in throughput compared to the host.

6.2.2 postgres

To understand the performance characteristics under a different workload, we deploy a `postgres` database on one node and benchmark it using `pgbench` [45] on another node. We benchmark the database with ten threads and 100 connections, and run the benchmark for 60 seconds. As before, we compare the performance of our eBPF fast-path to that of a standard Linux VXLAN overlay network and `postgres` running on the host. Figure 6.6 show the average latency across three repeats as well as the total number of transaction executed during the benchmark.

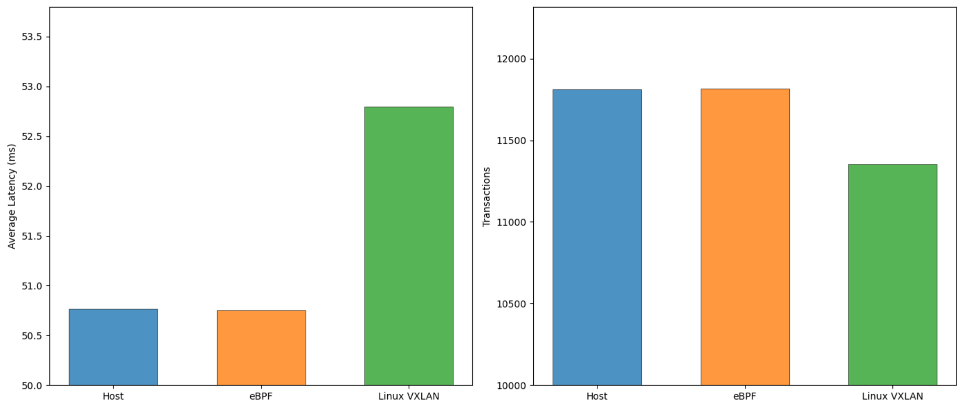


Figure 6.6: Comparison of latency (left) and number of transaction (right) when benchmarking a `postgres` database with `pgbench`. Note that the vertical axis do not start at zero.

The performance benefit of both our eBPF fast-path and the host is much less pronounced than in the case of `nginx` and the microbenchmarks, but still exists. The eBPF fast-path achieves only a 3.87% decrease in latency and a 4.05% increase in throughput compared to the Linux VXLAN overlay. This could be due to the fact that `postgres` typically uses smaller packet payloads in comparison to web servers, which means that the 50 byte overhead of the VXLAN header is less likely to cause any packet fragmentation. Additionally, `postgres` performs heavier user-space processing than `nginx` since it needs to parse, plan and execute SQL queries. The much higher average latencies in the case of `postgres` ($\approx$ 50000ms) compared to `nginx` ($\approx$ 700ms) further support this analysis.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we presented our design and implementation for a kernel space system call argument rewriting mechanism that uses eBPF. The primary advantage of this mechanism is that it does not require rewriting binaries to perform system call interception. This makes it suitable for use by cloud providers, who are unwilling to modify the binaries of their customers. We demonstrate the utility of these eBPF hooks by implementing a lightweight alternative to network namespaces. We use this to provide network isolation for Kubernetes pods placed in the host network namespace. This approach results in a 92× reduction when launching 200 containers in parallel and a 60% reduction in the launch time of a single container. Additionally, we achieve a 35% increase in throughput and a 33% reduction in latency for pod-to-pod communication when compared to the standard Linux VXLAN overlay implementation.

7.2 Limitations

The most significant limitation of this work is the fact that the eBPF programs are attached to the Kubernetes pod cgroup after the pod is created, rather than immediately after the cgroup is created and before any applications run inside the container. This is due to the fact that the BPF agent runs as a DaemonSet and listens for Kubernetes API events. The standard way to overcome this limitation is to instead create a custom container network interface (CNI) plugin that sets up pod networking and attaches the eBPF programs at cgroup creation time.

Another limitation is that the eBPF fast-path can only be used for a limited number of connections, since it uses the host network IP address. There are various ways to overcome this limitation, but each has its own trade-offs and solutions must be carefully chosen to match the use case. For example, cloud providers could attach multiple IPv6 addresses to the host network interface. This way, the range of ports that can be allocated to a single pod could significantly be increased. At the extreme, a single IPv6 address could be assigned per pod.

Finally, the eBPF system call interception mechanism currently only supports a limited set of system calls. Specifically, we implemented the system calls required for our use case of a lightweight alternative to network namespaces. However, extending the mechanism to support additional system calls is straightforward and involves applying the simple pattern established in chapter 4.

7.3 Future Work

While this thesis focused on network namespaces and overlay networks, system call interception with eBPF could potentially be used to implement other sandboxing mechanisms. Doing so may have further performance benefits. To further extend the power of our eBPF hooks, we could introduce a set of eBPF KFuncs. Currently, the eBPF programs aren't able to perform arbitrary operations such as invoking any other system call. User-space interposition mechanisms do not have this limitation. By exposing a set of KFuncs, we could allow eBPF programs to perform powerful kernel operations.

Since the kernel modifications required for the eBPF system call hooks are relatively small and well separated from the rest of the kernel, maintaining a kernel fork and applying a patch including the eBPF hooks is a viable option. However, it would be preferable to have the eBPF hooks included in the mainline kernel. This may prove to be challenging since the kernel community is generally rather sceptical of open-ended eBPF hooks.

Bibliography

[1] Katran. https://github.com/facebookincubator/katran, 2025. Accessed: 04-06-2025.

[2] Noisy Neighbour Detection. https://netflixtechblog.com/noisy-neighbor-detection-with-ebpf-64b1f4b3bbdd, 2025. Accessed: 04-06-2025.

[3] Firecracker. https://firecracker-microvm.github.io/, 2025. Accessed: 07-06-2025.

[4] Unimog. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/, 2025. Accessed: 04-06-2025.

[5] L4 Drop. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/, 2025. Accessed: 04-06-2025.

[6] Adriaan Jacobs, Merve Gülmez, Alicia Andries, Stijn Volckaert, and Alexios Voulimeneas. System call interposition without compromise. In *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 183–194, 2024.

[7] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, Boston, MA, July 2023. USENIX Association.

[8] Amazon Web Services. Shared Responsibility Model. https://aws.amazon.com/compliance/shared-responsibility-model/, 2025. Accessed: 10-06-2025.

[9] Yougang Song and Brett Fleisch. Utilizing binary rewriting for improving end-host security. *Parallel and Distributed Systems, IEEE Transactions on*, 18:1687–1699, 01 2008.

[10] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Seccomp. https://kubernetes.io/docs/tutorials/security/seccomp/, 2025. Accessed: 07-06-2025.

[12] Jim Keniston. Kernel Probes. https://docs.kernel.org/trace/kprobes.html, 2025. Accessed: 10-06-2025.

[13] Matthieu Desnoyers. Kernel Tracepoints. https://docs.kernel.org/trace/tracepoints.html, 2025. Accessed: 10-06-2025.

[14] Wine. https://www.winehq.org/, 2025. Accessed: 10-06-2025.

[15] Jeff Dike. User-mode linux. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ALS '01, page 2, USA, 2001. USENIX Association.

[16] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 16–29, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, page 189–197. IEEE Press, 2018.

[18] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a Low-Overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, Boston, MA, February 2019. USENIX Association.

[19] Shengkai Lin, Shizhen Zhao, Peirui Cao, Xinchi Han, Quan Tian, Wenfeng Liu, Qi Wu, Donghai Han, and Xinbing Wang. ONCache: A Cache-Based Low-Overhead container overlay network. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 979–998, Philadelphia, PA, April 2025. USENIX Association.

[20] eBPF Documentation. `https://docs.ebpf.io/`, 2025. Accessed: 04-06-2025.

[21] Cilium. `https://cilium.io/`, 2025. Accessed: 04-06-2025.

[22] eBPF Socket System Call Hooks. `https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_CGROUP_SOCK_ADDR/`, 2025. Accessed: 05-06-2025.

[23] eBPF Bind Hook. `https://github.com/torvalds/linux/commit/4fbac77d2d092b475dda9eea66da674369665427`, 2025. Accessed: 11-06-2025.

[24] Cncf, annual survey 2023. `https://www.cncf.io/reports/cncf-annual-survey-2023/`, 2023. Accessed: 02-06-2025.

[25] Nginx. `https://nginx.org/`, 2025. Accessed: 02-06-2025.

[26] Postgresql. `https://www.postgresql.org/`, 2025. Accessed: 02-06-2025.

[27] Redis. `https://redis.io/`, 2025. Accessed: 02-06-2025.

[28] Kubernetes overview. `https://kubernetes.io/docs/concepts/overview/`, 2025. Accessed: 02-06-2025.

[29] AWS Lambda. `https://aws.amazon.com/lambda/`, 2025. Accessed: 02-06-2025.

[30] Namespaces, Linux Manual. `https://man7.org/linux/man-pages/man7/namespaces.7.html`, 2025. Accessed: 02-06-2025.

[31] PID Namespaces, Linux Manual. `https://man7.org/linux/man-pages/man7/pid_namespaces.7.html`, 2025. Accessed: 02-06-2025.

[32] Control Groups, Linux Manual. `https://man7.org/linux/man-pages/man7/cgroups.7.html`, 2025. Accessed: 02-06-2025.

[33] Network Namespaces, Linux Manual. `https://man7.org/linux/man-pages/man7/network_namespaces.7.html`, 2025. Accessed: 02-06-2025.

[34] Virtual Ethernet Devices, Linux Manual. https://man7.org/linux/man-pages/man4/veth.4.html, 2025. Accessed: 02-06-2025.

[35] Virtual Bridges, Linux Manual. https://man7.org/linux/man-pages/man8/bridge.8.html, 2025. Accessed: 02-06-2025.

[36] Gourav Shah. Docker Documentation. https://docker-tutorial.schoolofdevops.com/swarm-networking-deepdive/, 2017. Accessed: 2025-01-19.

[37] et al Mahalingam. VXLAN. https://datatracker.ietf.org/doc/html/rfc7348, 2014. Accessed: 2025-01-19.

[38] Netlink, Linux Manual. https://man7.org/linux/man-pages/man7/netlink.7.html, 2025. Accessed: 03-06-2025.

[39] eBPF Go Library. https://ebpf-go.dev/, 2025. Accessed: 05-06-2025.

[40] iperf3. https://github.com/esnet/iperf, 2025. Accessed: 10-06-2025.

[41] Taskset, Linux Manual. https://www.man7.org/linux/man-pages/man1/taskset.1.html, 2025. Accessed: 10-06-2025.

[42] iproute2. https://github.com/iproute2/iproute2, 2025. Accessed: 10-06-2025.

[43] Understanding VXLAN+OVS Bandwidth Issues. https://www.stackhpc.com/vxlan-ovs-bandwidth.html#:~:text=A%20very%20simple%20test%20case,single%20TCP%20stream%20between%20them. Accessed: 08-06-2024.

[44] wrk. https://github.com/wg/wrk, 2025. Accessed: 10-06-2025.

[45] pgbench. https://www.postgresql.org/docs/current/pgbench.html, 2025. Accessed: 11-06-2025.

Chapter 8

Declarations

8.1 Use of Generative AI

Generative AI tools have been used as companions throughput this project, including ChatGPT and Claude. These tools have assisted in answering questions about the Linux kernel source code and the current landscape of container networking. Furthermore, they have been used to discuss ideas for the design and implementation of both the eBPF system call hooks and the lightweight alternative to network namespaces.

8.2 Ethical Considerations

There are no ethical considerations related to this project.

8.3 Sustainability

There are no sustainability considerations related to this project.

8.4 Availability of Data and Materials

All code is available on GitHub: `https://github.com/lucasbn/ebpf-overlay`