Technical Report

UCAM-CL-TR-1013
ISSN 1476-2986

Number 1013



UNIVERSITY OF CAMBRIDGE

Computer Laboratory

TraceNS: partitioned system tracing for container paradigms

Juntong Deng

November 2026

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 2026 Juntong Deng

This technical report is based on a dissertation submitted June 2026 by the author for the degree of Master of Philosophy (Advanced Computer Science) to the University of Cambridge, Hughes Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-1013*

Abstract

In the development and operation of modern software systems, many system behaviors become opaque and difficult to understand due to the increasing complexity of the software. System tracing is a fundamental method to help developers understand the behavior of complex systems, providing critical runtime information about the system. Effective, reliable, and ubiquitous system tracing is crucial for the analysis, debugging, and optimization of complex systems. However, with the adoption of novel container system environments, modern operating systems still offer limited support for system tracing in these environments. In multi-tenant container system environments, system tracing is not practically available to developers.

This study proposes TraceNS, a novel partitioned system tracing scheme designed for container paradigms. TraceNS enables developers to perform efficient and secure system tracing within containers in multi-tenant container system environments. Through case studies, this study demonstrates how TraceNS can help developers analyze, debug, and optimize their containers. According to the evaluation of TraceNS in this study, TraceNS achieves sufficient tracing coverage of container-related system events and security in multi-tenant container system environments, while maintaining a low overhead for container creation and runtime.

Acknowledgements

I am very grateful to my supervisor, Prof Andrew W. Moore, who provided important support and guidance for my project. I meet with my supervisor every week, and I thoroughly enjoy every conversation with him. My supervisor was interested in my project, which was a great encouragement. All of these have been very helpful to my project.

Contents

1 Introduction **9**

2 Background & Related Work **13**
2.1 Containers and Virtualization 13
2.2 Linux Namespaces and Cgroups 15
2.3 System Tracing in Operating Systems 16
2.4 Cloud Environment Observability 16
2.5 Container System Tracing 17

3 Design **19**
3.1 System Operation Partition Context 19
3.2 Operation Partition Context Establishment 20
3.3 Operation Partition Context Inheritance 22
3.4 Multi-instance Tracing in Isolated Views 22
3.5 Sensitive Information Hiding 24

4 Implementation **27**
4.1 Tracing Namespace 27
4.2 Ftrace Instance 28
4.3 Linux Kernel Execution Contexts 28
4.4 Establish Operation Partition Context 30
4.5 System Tracing Event Delivery 31
4.6 Tracing Cgroup Controller 32

5 Evaluation **33**
5.1 Performance 33
5.1.1 Container Creation 34
5.1.2 Container Runtime 37
5.2 System Event Coverage 40
5.2.1 System Call 42
5.2.2 Predefined System Events in Kernel Subsystems 42
5.3 Security 44
5.3.1 Event Isolation 44
5.3.2 Sensitive Information Hiding 46
5.4 Summary 47

6 Case Studies **49**
6.1 Eliminating Container Long-tail Latency 49
6.2 Container Network Troubleshooting 50
6.3 Solving a Container Quirk 52

6.4 Improving Container Performance 53
6.5 Summary 55

7 Conclusions & Future Work **57**
7.1 Conclusions 57
7.2 Future Work 58
7.2.1 Multi-user Environment Tracing (Non-container) 58
7.2.2 Integration with Novel Tracing Technologies 58
7.2.3 Predefined Events and Kernel Function Coverage 59
7.2.4 Noisy Neighbors in Cloud Environments 59
7.2.5 Distributed, Large-scale, High-density Environments 60

Chapter 1

Introduction

System tracing is a crucial infrastructure in modern operating systems, enabling developers to analyze the behavior of complex systems and applications by capturing and logging runtime information (events) [3, 15]. For troubleshooting and performance optimization of complex systems, system tracing can show whether the system is operating as expected, and many case studies in the industry demonstrate that system tracing plays a crucial role [36, 38, 5, 12, 95, 94]. In academic research, systems tracing can provide a deep understanding of the behavior of systems, reveal imperceptible properties within systems, offer counterintuitive new insights, and help scientists and engineers design better systems [83, 14, 6, 91, 100, 98, 39, 11, 10].

System tracing has been widely used in traditional bare-metal and virtual machine environments. However, in some more recent novel system environments, such as container environments, system tracing faces significant challenges. Containers are a lightweight virtualization technology that achieves virtualization at the operating system level, without requiring hardware simulation or a separate operating system kernel [93]. Containers are based on the resource partitioning mechanism in the operating system, providing different isolated resource views for different containers (processes) to provide isolated virtual system environments [80]. Compared to traditional hardware virtual machines, containers require fewer resources and offer better performance [8, 43], leading to containers widely used in the cloud computing industry [99].

However, operating system-level virtualization presents new significant challenges for system tracing, especially in multi-tenant cloud environments. All containers share the same underlying operating system kernel [93, 80], as shown in Figure 1.1, but system tracing is global (system-wide). Modern operating systems only support filtering of system tracing events, but cannot truly achieve system event partitioning, and cannot actually provide different isolated event views for different processes [23, 79]. The lack of isolation could lead to security issues. For example, in multi-tenant environments, if cloud provider offers a global view of system events within containers, then tenants will be able to see all system events (including those of other tenants), overloading every container with unnecessary information and leading to the leakage of private information across tenants.

The lack of practically usable system tracing infrastructure for containers is a critical gap. Tenants are unable to analyze system-level performance issues and anomalies within their containers through system tracing. For example, through system tracing, when database performance degrades, the system behaviors causing the performance degradation can be quickly identified [36], and when the network loses packets for no apparent reason,

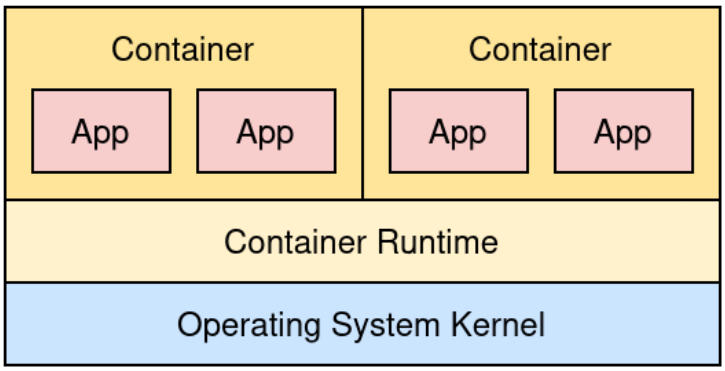

Figure 1.1: Container architecture. All containers share the same operating system kernel.

the kernel function that triggered the packet loss in the system can be quickly located [95]. In current cloud environments, tenants cannot analyze system-level issues and need to request assistance from cloud providers to resolve these issues, which is inconvenient and inefficient for tenants. Tenants and cloud providers need a secure system tracing infrastructure that is available in container environments.

System tracing events originate from system operations, and modern operating systems cannot achieve system event partitioning because they cannot achieve system operation partitioning. Compared to the already achieved system resource partitioning [61], system operation partitioning is a more difficult challenge. Associating system resources (such as network interfaces [62]) with a specific set of processes is straightforward. However, this is not the case for system operation. System operation can be a complex and dynamic process. For example, in packet arrival operations, packets may cross multiple network partitions (packets are routed across multiple container networks), and the process to which the packet will arrive is not known until the packet is demultiplexed. Before that, it is challenging to associate the packet with a set of processes.

In addition to isolated event partitioning, the system tracing infrastructure available within containers faces other difficult challenges. The tracing subsystem is one of the most performance-critical kernel subsystems. System tracing must be very lightweight, because instrumentation can occur in various operations within the system, including operations on performance-sensitive hot paths. Excessive tracing overhead can affect the accuracy of system and application performance measurements [34].

Furthermore, the tracing logs provided to containers must not contain sensitive information that could potentially compromise system security, such as information that might reveal the kernel memory layout. The system tracing available within containers must cover enough container-related system events to help developers better understand container behavior in multi-tenant container environments.

This study proposes TraceNS, bridging the critical gap where modern operating systems are unable to provide effective system tracing in multi-tenant container environments. TraceNS achieves system operation partitioning by creating partition contexts for system operations in the operating system kernel, thereby enabling system tracing event partitioning. TraceNS provides isolated and secure system tracing views for different containers through multi-instance system tracing associated with partition context.

The prototype implementation of TraceNS integrates with current container technologies (namespaces [61], cgroups [55]) and system tracing infrastructure (Ftrace [23]). This enables users to use familiar system interfaces and leverage existing tracing toolchains

(such as trace-cmd [68], kernelshark [47], perf [85]). Users can create system tracing partitions without using special system interfaces, and capture complex container behaviors within containers for visualization, performance analysis, container interaction analysis, and more without using special tracing tools.

In the evaluation of this study, TraceNS has sufficient coverage for container-related system events, achieves secure system event isolation, and provides options for hiding sensitive information in tracing logs. TraceNS has low runtime tracing overhead through minimized hot path operations and low container creation overhead through lazy initialization. Finally, this study demonstrates through case studies that TraceNS can effectively help developers enhance their understanding of container behavior and assist them in analyzing, debugging, and optimizing containers.

This study presents TraceNS, the first approach to provide secure and efficient system tracing infrastructure within containers. The main contributions of this study are:

- This study describes the background and related works of TraceNS, and discusses the limitations of existing works (§2).
- This study describes the design of TraceNS, the first design to establish partition context for system operations and provide isolated multi-view system tracing (§3).
- This study describes the prototype implementation of TraceNS, the first available secure and efficient in-container system tracing infrastructure (§4).
- This study proposes the first evaluation methodology for such partitioned system tracing infrastructures, including performance, system event coverage, and security, and provides a comprehensive evaluation of TraceNS (§5).
- This study provides case studies to demonstrate how TraceNS can help developers analyze, debug, and optimize containers in real-world scenarios (§6).

Chapter 2

Background & Related Work

This chapter describes the background and related work of TraceNS, including containers and virtualization (§2.1), Linux namespaces and cgroups (§2.2), system tracing in operating systems (§2.3), cloud environment observability (§2.4), and container system tracing (§2.5). Containers (operating system-level virtualization) is the main usage scenario for TraceNS. Namespaces and cgroups provided by the Linux kernel are the key technologies upon which Linux containers rely. Cloud observability is the scenarios TraceNS aims to enhance, where system-level observability faces limitations that prevent it from being available in multi-tenant container environments. The gaps of container system tracing are what TraceNS aims to bridge, as limitations in existing work have resulted in the lack of a truly secure and effective solution.

2.1 Containers and Virtualization

In modern operating systems and cloud computing industries, the commonly used methods for creating isolated virtual system environments are hardware-level virtualization and operating system-level virtualization. Figure 2.1 illustrates the architecture of different virtualization technologies.

Bare metal is the simplest case, and it is not virtualization. All applications run directly on the operating system. The operating system runs directly on the hardware. Applications can see each other and can use all resources on the system. This situation is common on (non-shared) internal servers.

In hardware-level virtualization, virtual machines provide virtual hardware environments by simulating hardware and manage hardware resources through a hypervisor, such as Xen [7] and KVM [48]. Each hardware virtual machine needs to build a complete system environment, run a separate operating system kernel, and run on the virtual hardware. Hardware-level virtualization inevitably incurs significant overhead during virtual machine startup and runtime.

Operating system-level virtualization is based on the system resource partitioning mechanism provided by the operating system kernel. Operating system-level virtualization divides system resources such as CPU, memory, network, and disk, and creates isolated virtual environments for processes [80]. The processes in a virtual environment can only see the resources of its own partition and can only use the amount of resources available in that partition. Operating system-level virtualization does not require simulating

hardware or providing a separate operating system kernel, thus having lower overhead compared to hardware-level virtualization [8, 43].

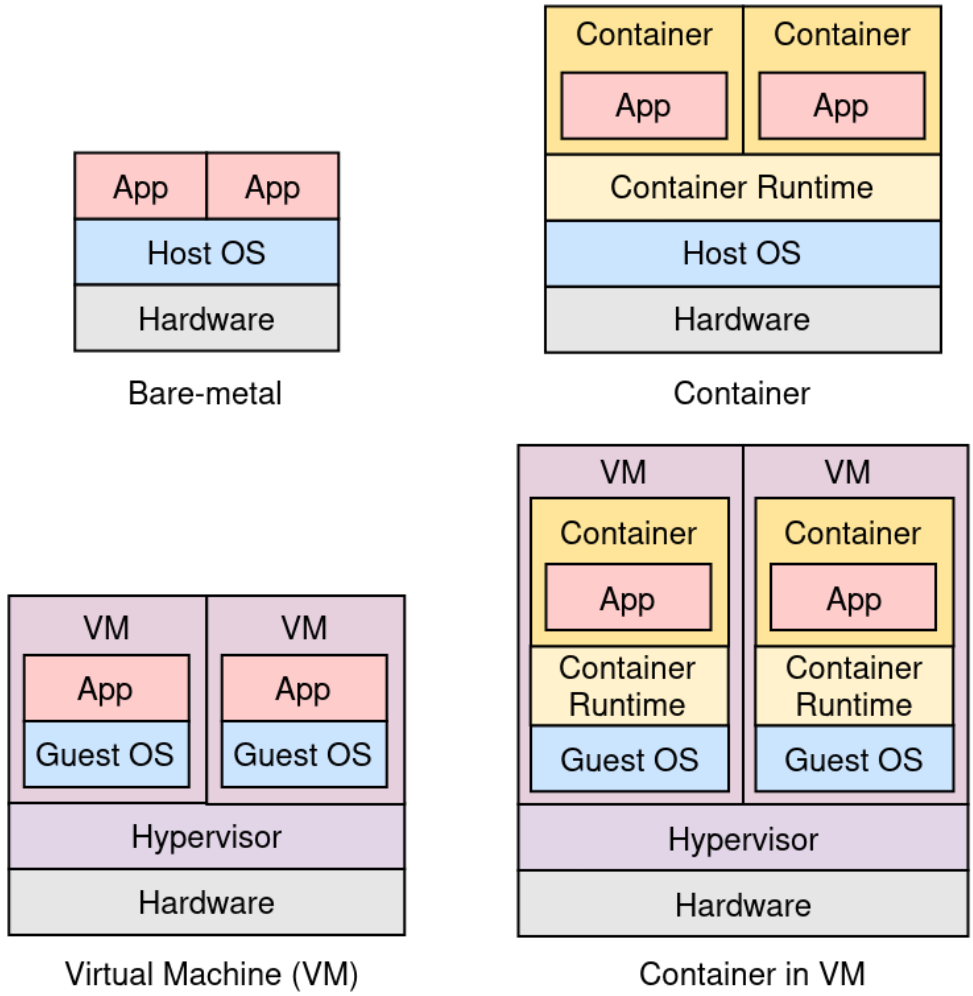

Figure 2.1: Comparison of architectures of different virtualization technologies, including bare metal (no virtualization), containers (operating system-level virtualization), virtual machines (hardware-level virtualization), and containers within virtual machines.

Operating system-level virtualization has different implementations on different operating systems, such as containers [93] in Linux, jails [44] in FreeBSD, and zones [89] in Solaris. Operating system-level virtualization is widely used in the cloud computing industry due to its low resource overhead [99]. Linux containers are currently the most widely used operating system-level virtualization technology. Linux containers create isolated virtual environments based on a series of system features provided by the Linux kernel, such as namespaces [61] (which restrict the container process view), cgroups [55] (which restrict the amount of resources available to the container process), capabilities [54] (which restrict the container process permissions), and seccomp [65] (which restricts the system calls available to the container process).

Linux containers are typically created and managed by container runtimes, and there are high-level container runtimes and low-level container runtimes [30]. High-level container runtimes are responsible for container cluster management, container network management, container image management, etc., such as Docker [22], Kubernetes [50], and Podman [87]. Low-level container runtimes are responsible for creating isolated container virtual environments, managing container lifecycles, etc., such as runc [92], crun [20],

and youki [102]. High-level container runtimes and low-level container runtimes collaborate through operation interfaces that conforms to the OCI (Open Container Initiative) runtime specification [40]. Low-level container runtimes provide specification-compliant operation interfaces, and high-level container runtimes operate low-level container runtimes through these specification-compliant interfaces.

To enable containers to have higher security (hardware-level isolation) and continue to benefit from the container ecosystem (images, orchestration tools, etc.), some container runtimes run containers in virtual machines (such as Kata Containers [46]). These container runtimes provide the same OCI operation interfaces as regular containers, but the containers run in lightweight virtual machines (such as Firecracker [1]). However, these technologies still have higher performance overhead (such as longer container startup time, I/O performance degradation due to virtual hardware [51, 18]) and more limited compatibility (such as limited hardware devices supported). These technologies have already been applied in the industry, such as AWS Lambda [2] and some scenarios with high security requirements, but in most common scenarios, regular containers are still the default choice for production environments.

2.2 Linux Namespaces and Cgroups

Linux namespaces [61] and Linux cgroups [55] provided by the Linux kernel are core technologies of Linux containers.

Linux namespaces restrict the resources that processes can access, enabling the isolation of resources accessible to processes and the creation of container virtual environments. Processes can only access resources within the namespaces they are located in and cannot access resources within other namespaces. Multiple processes can exist in one namespace, and a resource may exist in multiple namespaces. There are currently eight types of namespaces available in the Linux kernel: Cgroup, IPC, Network, Mount, PID, Time, User, and UTS. Initially, only the root namespace existed in the system. Processes can create new namespaces using the `unshare()`, `clone()`, and `clone3()` system calls [69, 56], or move to another (existing) namespace using the `setns()` system call [66]. In this study, TraceNS provides the mechanism for creating system tracing partitions to userspace through namespaces (tracing namespace).

Linux cgroups (control groups) restrict the amount of resources that processes can use, preventing the container from consuming too many system resources. Users can use cgroups to impose restrictions on various types of system resources, such as CPU, disk I/O, memory, network, and devices. The cgroups interface is provided through the cgroup pseudo-filesystem, which allows users to define hierarchical structures and attributes, and bind a set of limits or parameters to a group of processes. In the cgroup pseudo-filesystem, users can create directories to create new control groups and create hierarchical structures to further subdivide resource usage limits. By reading and writing files in directories, users can add processes to a control group or set usage limits for various resources in a control group. Cgroups has two versions of the interface, V1 and V2, and the V1 version interface is about to be deprecated. In this study, TraceNS provides options via cgroup (tracing cgroup controller) to restrict the types of sensitive information in tracing logs that processes within containers can view.

2.3 System Tracing in Operating Systems

Modern operating systems typically provide system tracing infrastructure to help users analyze the behavior of systems and applications, such as Ftrace [23] in Linux and Dtrace [15] in FreeBSD and Solaris. Since TraceNS is based on Linux kernel, this section focuses on the system tracing infrastructure in Linux systems.

The Linux kernel provides extensive system tracing infrastructure that enables users to trace system events and kernel function calls [25]. The Linux kernel predefines over a thousand static tracepoints as traceable system events, and provides a function tracer that allows tracing over 70,000 kernel functions, enabling users to understand what is happening in over a hundred kernel subsystems. In addition to predefined events and kernel functions, the Linux kernel provides kprobe/uprobe, which enables users to instrument and trace any location in the kernel and userspace applications [24, 27]. The tracing interface provided by the Linux kernel is Ftrace, which is exposed as a pseudo-filesystem, allowing users to operate and configure the tracing infrastructure by reading/writing files. For example, users can set the kernel functions to be traced using the `set_ftrace_filter` file, set the system events to be traced using `set_event` file, and read the tracing logs using `trace_pipe` file, and so on.

In recent years, the emergence of eBPF [29] technology has further enhanced the system tracing infrastructure of Linux systems and made system tracing more flexible and powerful. eBPF achieves secure dynamic extensions to the kernel, allowing hooks to be safely added for tracing in almost any location within the kernel without requiring kernel recompilation or system reboot, and eBPF programs have low overhead after JIT compilation. The Linux kernel supports attaching eBPF programs to static tracepoints, or attaching eBPF programs to the begin/end of kernel functions via BPF Fentry/Fexit for system tracing [28]. Users can write their own native eBPF programs for system tracing or use high-level tracing tools such as bpftrace [13].

However, whether it is Ftrace or eBPF in Linux, or tracing infrastructure in other operating systems, system tracing is system-wide (global) and can trace all events across the entire system. Therefore, system tracing is typically only allowed to be used by system administrators, and in multi-tenant container environments, system tracing cannot be used in containers, otherwise it would cause security issues such as leakage of privacy information across containers.

2.4 Cloud Environment Observability

With the rapid development of the cloud computing industry, the workloads that tenants run on cloud environments are becoming increasingly complex, to help tenants and cloud providers understand what is happening in the increasingly complex cloud environment, observability in the cloud environment has become an active area [49].

Nowadays, cloud providers typically offer some observability services that allow tenants to understand and monitor the behavior, health, and performance of their services. Observability in the cloud environment is provided through telemetry data, including metrics, logs, traces, and other information [17]. If tenants are not satisfied with the observability information provided by the cloud provider, they can deploy their own observability frameworks, such as OpenTelemetry [82], Prometheus [90], and Grafan [35], and collect telemetry data themselves. Many cloud-native applications provide data formats and

APIs that conform to these observability frameworks. All of these are mature solutions in the cloud computing industry.

For system-level issues, application-level observability is insufficient; system-level observability is required. With the emergence of novel tracing technologies such as eBPF [29], system-level observability in cloud environments has been further improved. Cilium [16], Parca [84], and Pixie [86] are commonly used system-level observability frameworks in cloud environments. Based on eBPF, they can help users collect system-level observability information. System-level observability of cloud environments based on eBPF has been currently widely used in the industry. In addition to regular observability, eBPF is also used for security enhancement in cloud environments. For example, Falco [31], Tetragn [96], and Calico [97] are cloud-native runtime security frameworks based on eBPF.

However, since the system tracing infrastructure such as eBPF on which system-level observability frameworks rely is still system-wide, currently only large technology companies or cloud providers that have their own cloud infrastructure can securely deploy system-level observability frameworks. Cloud providers that allow tenants to use system-level observability frameworks in multi-tenant container environments can cause the same security issues as described in §2.3. For small and medium-sized enterprises or individual developers who are container tenants, they cannot deploy their own system-level observability framework in containers. In this case, tenants need to request assistance from cloud providers to collect system-level observability information, which is inconvenient and inefficient.

2.5 Container System Tracing

System tracing in container environments has been a long-standing need, and many people have discussed and tried to provide solutions to better trace containers. However, there is still no available solution to provide secure and efficient system tracing within containers. Even outside of containers, current solutions for container tracing still have inherent limitations. This section describes related works and their limitations.

Jeanson proposed an event filtering method that supports using process's namespaces [61] as context (mainly PID namespaces [64]) to achieve better container tracing event filtering [41, 42]. Craun et al. proposed that when using eBPF for tracing, untraced (uninterested) processes (containers) should be filtered based on their PIDs before actually processing events, in order to reduce the tracing overhead caused by untraced processes [19]. Karadzhov pointed out that tracing container workloads is an attractive idea and discussed it, but did not provide a solution [45]. Desnoyers and Jeanson preliminary discussed about the idea of tracer namespaces for filtering and partitioning system tracing events, but they did not provide any detailed methods or solutions [21].

Existing works aim to achieve better filtering of container tracing events outside the container. However, even outside of containers, filtering methods based mainly on PID (or PID namespace) have inherent limitations. Many container-related system operations in the kernel do not occur in the context of the container processes, but rather in the context of interrupts or kernel threads (such as network packet receiving or disk data cache write-back). System events in these system operations cannot be filtered based on the PIDs of container processes.

For scenarios within containers, filtering system tracing events as they are processed, rather than isolating them when they are triggered, inherently unable to provide a truly

isolated tracing view and does not help enable secure system tracing for tenants in multi-tenant container environments. In multi-tenant environments, it is not secure enough to help tenants ignore (filter out) events that do not belong to them when they are processing system tracing events. Security can only be guaranteed if tenants are not delivered with system tracing events that do not belong to them in the first place.

Chapter 3

Design

The main design goal of TraceNS is to establish partition contexts for system operations so that the current partition can be identified when relevant system events are triggered, and to provide system tracing interfaces with isolated views to user space. This chapter describes the high-level design principles of TraceNS. The specific implementation details may vary on different operating systems. The implementation of TraceNS depends on the characteristics of the specific operating system in execution contexts, operating system-level virtualization mechanism, and system tracing infrastructure.

3.1 System Operation Partition Context

All system events originate from system operations. In order to achieve system event partitioning, a key prerequisite is to achieve system operation partitioning, that is, to have a mechanism to identify which partition a system operation belongs to. The approach used in this study is to establish an operation-global partition context for a system operation, which is effective throughout the entire system operation process, as shown in Figure 3.1. A system operation can call a series of kernel functions, and in fact, triggering an system event is also a kernel function call. Only the operation-global context of a system operation is available in all functions called (the entire process) by the system operation. The partition context is stored in different locations for different system operation types, which will be described in the implementation chapter (§4).

Another approach that seems feasible but is not practically feasible is to explore event parameters when a system event is triggered, and attempt to deduce the partition in which the current system operation is located based on the information in the event parameters. However, not every system event has enough information in event parameters to deduce the current partition. In addition, the parameters for each system event vary widely, so this approach is not scalable.

The approach of the operation-global partition context chosen in this study is a scalable approach. The operation-global partition context enables the identification of partitions of all system events in a system operation, where a system operation can trigger any number of system events, without requiring the understanding or parsing of the parameters for each system event.

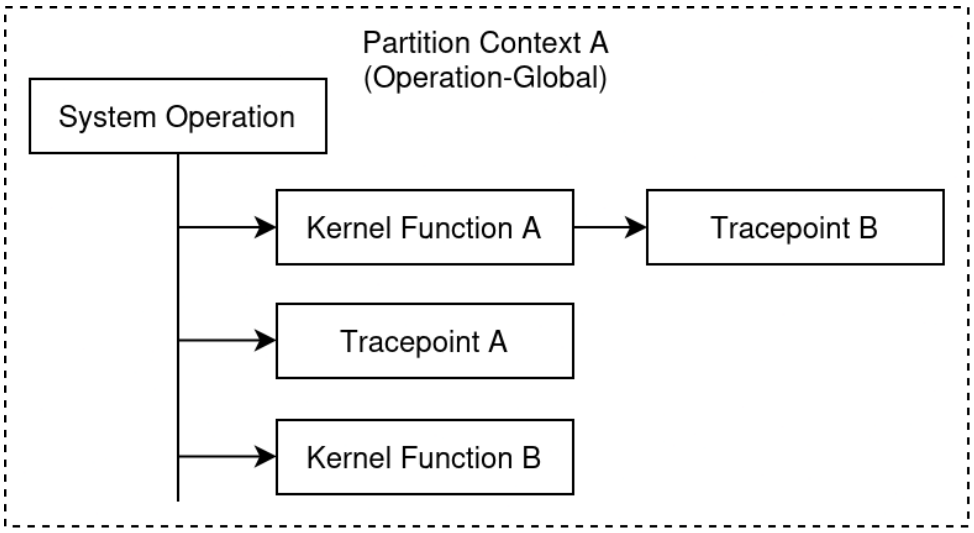

Figure 3.1: The operation-global system partition context is available throughout a system operation and can be accessed in all functions called and all system events triggered during the system operation.

3.2 Operation Partition Context Establishment

As an infrastructure that manages and abstracts hardware and provides common services to applications, most system operations in operating systems originate from external calls, such as system calls from userspace processes or interrupts from underlying hardware. From another practical perspective, only system operations with a clear source can be partitioned. Therefore, the appropriate time to establish partition contexts is when system operations are initiated. At this point, the system operation is still at the boundary of the operating system kernel (or kernel subsystem), rather than deep within the operating system kernel, and the source operation partition is relatively easy to identify. Unlike system resources, system operations are not an entity, but a dynamic "action". The operation partitions for system operations are not naturally present and need to be identified and established as required.

The approach used in this study is to add "checkpoints" at the boundaries of the operating system kernel and kernel subsystems as additional border control, as shown in Figure 3.2, such as at the entry point of interrupt handling or system call. When a system operation passes through these checkpoints, the operation partition context of the current system operation is established based on the current source of the system operation. Some system operations may traverse multiple operation partitions, so the current operation partition context will be updated at the boundary of each operation partition (usually the boundary of the kernel subsystem), as shown in Figure 3.3. Once a system operation is completed, the current partition context will be cleared to avoid affecting subsequent system operations.

In modern operating system kernels, there are many subsystems responsible for different functions, and the system operations in different subsystems have different characteristics. System operations that cannot be identified as belonging to a specific operation partition (where the source cannot be partitioned, such as the power subsystem) will not have an operation partition context established. Therefore, the events in these system operations can only be seen in the global system tracing interface (outside the container).

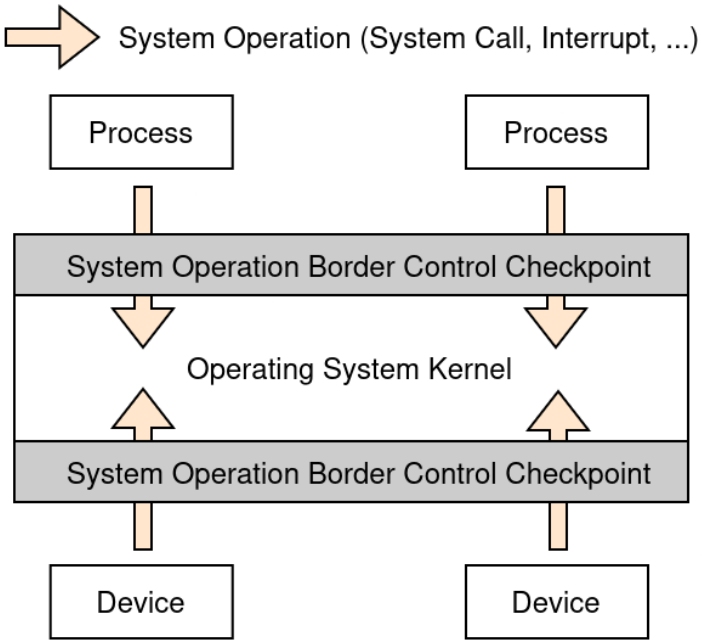

Figure 3.2: Border control checkpoints are added at the boundaries of the operating system kernel to attempt to establish operation partition contexts for system operations originating from external (such as system calls, interrupts, etc.).

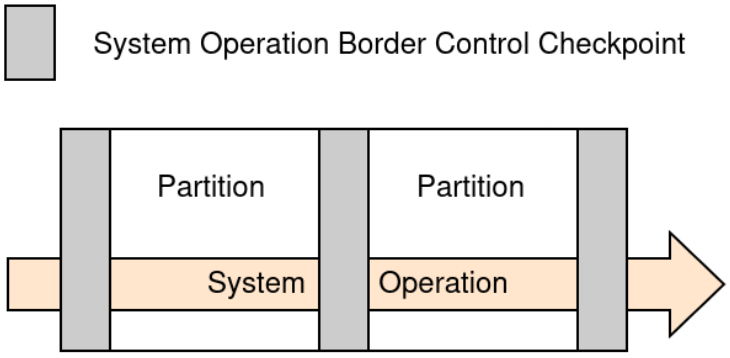

Figure 3.3: Border control checkpoints are added at the boundaries of the operating partition to attempt to establish/update operation partition contexts for system operations that cross the operating partition.

3.3 Operation Partition Context Inheritance

Not all system operations originate directly from external calls (such as system calls or interrupts); in some cases, they are indirect, with one system operation originating from another. For example, a system operation might create a timer to execute a task at a set time, add a task to a public task work queue, or create a new thread to execute the task. These will generate new system operations and execute them in a new context, but these new system operations should still belong to the partition where the original system operations were located.

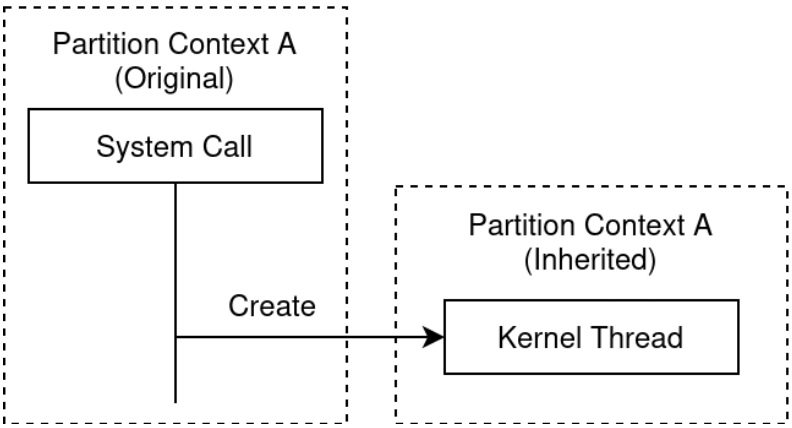

Figure 3.4: An example of partition context inheritance. A kernel thread inherits the partition context from its creator (system call).

The approach used in this study is that child system operations (such as timers, threads, work queues) inherit the operation partition context of the parent system operation (originating from an external call), as shown in Figure 3.4. System events triggered by child system operations can be seen in the system tracing interface of the operation partition context where the parent system operation were located. For example, if an interrupt places a complex task into a work queue for deferred execution, then the system events triggered when the task is executed in the work queue will be visible in the system tracing interface of the operation partition to which the interrupt belongs. Another example of partition context inheritance is that TCP sockets internally create a retransmission timer when the application creates a TCP socket via the `socket()` system call [67]. When the retransmission timer times out, the retransmission event will be visible in the system tracing interface of the operation partition to which the `socket()` system call belongs.

3.4 Multi-instance Tracing in Isolated Views

In modern operating systems, system tracing is global. In container environments, even if multiple system tracing instances are created (such as multiple eBPF programs attached to different system events), each tracing instance can still see system events triggered by different containers across the entire system, rather than only seeing system events from a specific container. In multi-tenant container environments, to ensure security, it is necessary to isolate the system tracing view of tracing instances. Each tracing instance can only see (delivered) system tracing events belonging to its own operation partition and cannot see system tracing events in other operation partitions. An example is given in Figure 3.5.

The approach used in this study is to add a system tracing event router that delivers

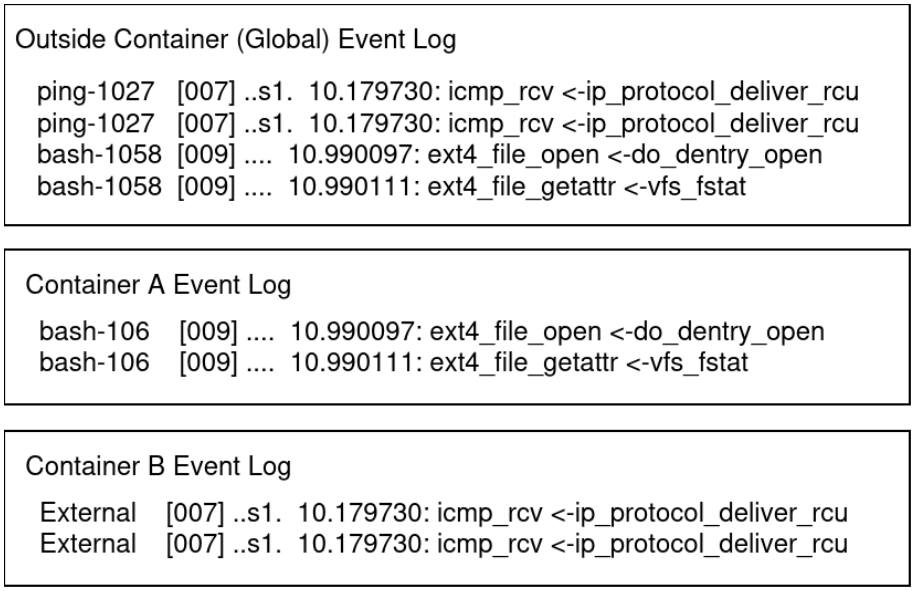

Figure 3.5: An example of isolated views. The global tracing instance is outside the container and can see all system events, but the tracing instances inside containers can only see its own system events. In container A, the process PID only shows the virtual PID within the container environment. In container B, system events originating from processes outside the container (in this example, ICMP requests sent by other containers) do not display the process name and PID, but only show "External".

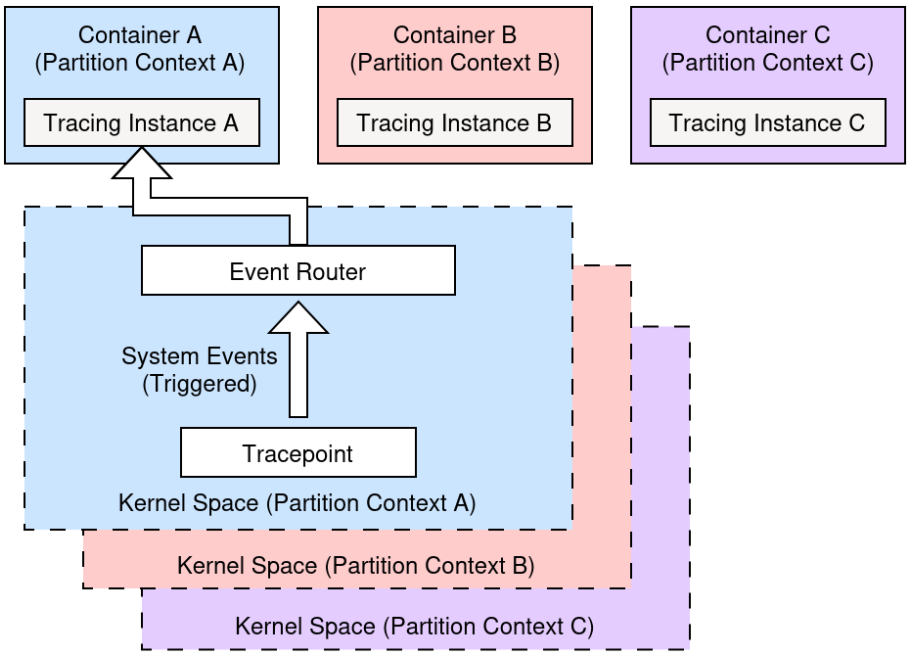

Figure 3.6: A system may contain multiple partition contexts (A, B, C), each corresponding to a different container (A, B, C). In this example, an system event triggered in partition context A will only be delivered to the tracing instance A of container A by the event router, thus achieving an isolated view.

system events only to the corresponding tracing instance based on the context of the operation partition in which the system event was triggered, as shown in Figure 3.6. In addition, the system tracing event router will deliver system events in all operation partitions to the global tracing instance without discrimination. The global system tracing instance should be able to see all system events, although further filters can be set to ignore system events that are not of interest. The global tracing instance is located outside the container and can only be accessed by system administrators. This design is based on the principle that container processes are just ordinary processes outside the container and container processes can be managed by system administrators in the usual way. The container only restricts the processes inside the container from viewing outside, but does not restrict external (system administrators) from viewing inside the container.

3.5 Sensitive Information Hiding

The system tracing event parameters may contain some sensitive information, such as the address of a kernel object, the offset of the specific kernel function that triggered the system event, and buffer data in the kernel, etc. This information could potentially leak the memory layout or contents of the current kernel. Although in most cases attackers (untrusted tenants) cannot directly use this sensitive information to attack the current system, when attackers discover vulnerabilities in the current system, this sensitive information may help attackers more easily exploit those vulnerabilities. Therefore, sensitive information that may exist in the system tracing logs needs to be appropriately hidden (covered up) according to the actual scenario. An example is given in Figure 3.7.

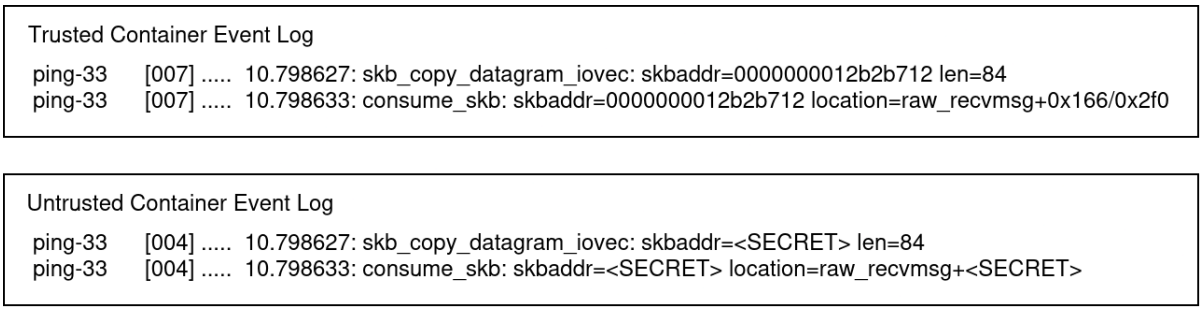

Figure 3.7: An example of sensitive information hiding. In a trusted container, all sensitive information is configured to be displayed. In an untrusted container, sensitive information is configured to be hidden, including object addresses and the specific offset of the kernel function that triggered the event.

The approach used in this study is to add a set of switches that enabled container creators to decide what system event log content the tracing instance should display to the processes inside the container. This study did not choose to hide all sensitive information directly, but instead left the decision to the users. In fact, this sensitive information can help diagnose issues. For example, kernel object addresses can help users identify objects between system events, and the function offsets that trigger system events can help users identify the problematic lines of code, and so on. A good design should provide mechanisms that allow users to choose, rather than presumptuously making choices for them (the principle of separation of mechanism and policy). Users can make different choices in different scenarios. For example, a cloud provider may allow more system event sensitive information to be displayed to help analyze issues when providing containers to internal employees or long-term partners (trusted), but may choose to hide all system

event sensitive information when providing containers to tenants on free tiers from the internet (untrusted).

Chapter 4

Implementation

This chapter describes the prototype implementation of TraceNS. This study chose to implement a prototype of TraceNS based on the Linux kernel (version 6.19.0). Linux is the operating system with the most comprehensive support for containers and is the most widely used operating system for deploying containers. The implementation of TraceNS described in this section is based on the characteristics of the Linux kernel in execution contexts, container-related technologies (operating system-level virtualization), and Ftrace [23] (system tracing infrastructure).

4.1 Tracing Namespace

To integrate with the existing Linux container technologies, this study extended the existing namespace technology in the Linux kernel and implemented the tracing namespace as an interface for userspace processes to operate on the tracing partition. Similar to other namespace types, userspace processes can create and enter a new tracing namespace using the `unshare()` and `clone3()` system calls, or enter an existing tracing namespace using the `setns()` system call [69, 56, 66]. Since the flags for namespace types in the `clone()` system call are full, new namespace types can only be created via the `clone3()` system call (the modern version of `clone()`). By default, child processes inherit all namespace types from their parent processes, and new namespaces are only created when the user specifies that a new namespace needs to be created.

In the implementation of this study, the tracing namespace is used as the partition context in §3.1 for system operations. When a tracing namespace is created, it records all other types of namespaces where the creator process resides in, using them as the scope (boundaries) of the current partition. This means that when creating a container, the tracing namespace must be the last namespace type created. Userspace programs normally do not need to pay special attention to this, because they usually use the `unshare()` or `clone3()` system calls to create and enter multiple types of namespaces at once, while in the tracing namespace implementation of this study, the tracing namespace is always created last in `unshare()` and `clone3()` system calls. However, if a userspace program chooses to create and enter only one namespace type at a time, then it needs to follow the specific order in which the tracing namespaces are last created.

4.2 Ftrace Instance

In the implementation of this study, the study provides isolated multi-instance tracing views for different containers in §3.4 through Ftrace instances [23]. Ftrace instances are provided through pseudo file system interfaces. Ftrace instances can be created and deleted as directories (using the `mkdir()` and `rmdir()` system calls [57, 58]), and accessed and configured through virtual files in the Ftrace instance directory. When an Ftrace instance is created, the tracing namespace of the process that created the Ftrace instance will be recorded, and the Ftrace instance will only be able to see system events in this tracing namespace (only events in this tracing namespace will be delivered).

If the container creator believes that the container may potentially use system tracing, then the container creator can create an Ftrace instance when the container is created and mount the Ftrace instance inside the container using a combination of the `open_tree()` and `move_mount()` system calls [63, 60]. This is a secure method provided by the Linux kernel for mounting external directories inside containers (in the mount namespace of the container). All created Ftrace instance directories can still be accessed and configured outside containers. This implementation follows the design principle that containers only restrict processes within the container but remain transparent to the outside world, and enables system administrators to manage the Ftrace instances running in containers.

Creating an Ftrace instance requires non-trivial initialization time, including allocating a system event ring buffer, initializing related data structures, and creating a series of virtual pseudo-filesystem files in the Ftrace instance directory as operation interfaces. To avoid the negative impact of creating Ftrace instances on container startup time, this study uses lazy initialization as an optimization in the implementation. When the process that creates the Ftrace instance is not in the root tracing namespace (which is usually for creating a container), it only creates an `activate` file in the Ftrace instance directory as an activation switch. No other initialization will be performed, no system event buffer will be allocated, no related data structures will be initialized, and no other virtual files will be created. The process within the container needs to activate the Ftrace instance by writing `1` to the `activate` file. Lazy-initialized Ftrace instances are only truly initialized when the Ftrace instance is activated.

Ftrace instance lazy initialization mechanism is based on the fact that most containers do not use system tracing during their lifecycle. It is only needed in a small number of cases when container tenants need to analyze system-level issues of the container. Therefore, most containers should not incur unnecessary Ftrace instance initialization overhead during their lifecycle and should not have their container startup time affected.

4.3 Linux Kernel Execution Contexts

In the Linux kernel, there are multiple execution contexts, each with different characteristics. This section describes these execution contexts and which execution contexts are relevant to this study or not. Table 4.1 lists the execution contexts and their characteristics in the Linux kernel that need to be described in this study. Table 4.2 lists the execution contexts relevant to the implementation of this study and examples that would occur within these execution contexts.

Execution Context	Task Context	Preemptible	Sleepable
Process	Yes	Yes	Yes
Kernel Thread	Yes	Yes	Yes
Hard Interrupt	No	Limited	No
Soft Interrupt	No	Limited	No
NMI	No	No	No

Table 4.1: Linux kernel execution contexts and their characteristics.

Execution Context	Examples
Process	system call
Kernel Thread	kthread, kworker, ksoftirqd
Soft Interrupt	softirq, tasklet, timer

Table 4.2: Linux kernel execution contexts related to this study and their examples.

Process

Process execution context is the context in which a userspace process executes system calls after entering the kernel. Each process execution context has its own task context. Process execution context is preemptible; it can be preempted by interrupts or by other processes during scheduling. Process execution context can sleep.

Kernel Thread

Kernel thread is a special type of "process" used to perform internal tasks of the kernel. Each kernel thread execution context has its own task context. Kernel thread execution context is preemptible; it can be preempted by interrupts or by other processes (including other kernel threads) during scheduling. Kernel thread execution context can sleep. Kernel threads can be created directly to execute tasks, or used indirectly by adding tasks to kernel workqueues. The workqueue tasks will be executed in the kworker kernel threads. The Linux kernel creates a ksoftirqd kernel thread on each CPU to handle situations where there is a significant backlog of soft interrupt tasks, and processes soft interrupt tasks that could not be completed in the soft interrupt within ksoftirqd kernel threads.

Hard Interrupt

Hard interrupt execution context is used to handle interrupt requests from the hardware. Hard interrupt execution context does not have its own task context, cannot sleep, and can be preempted by higher-priority hard interrupts and NMI. In the Linux kernel, hard interrupts are executed briefly and are used only for the minimum necessary work. Most of the work will be deferred to soft interrupts or work queues as the "bottom half". The relevance of hard interrupts to containers is limited, therefore this study will not establish partition contexts for hard interrupt execution contexts.

Soft Interrupt

Soft interrupt is a deferred execution mechanism used to handle more complex tasks that cannot be completed during the brief execution of hard interrupts. Soft interrupt execution context does not have its own task context, cannot sleep, and can be preempted by hard interrupts and NMI, but cannot be preempted by another soft interrupt. Tasklets

and timers in the Linux kernel are implemented based on soft interrupts. Tasklet is a high-level mechanism that is used to perform deferred execution tasks and is easier to use than low-level soft interrupts. Although soft interrupts are designed to handle more complex tasks, they can only be executed for a limited time. When there are too many soft interrupt tasks, the backlog of tasks will be processed in ksoftirqd kernel threads.

NMI

NMI execution context is used to handle non-maskable interrupts (NMI). NMI does not have its own task context, cannot sleep, and cannot be preempted. NMI is used only to perform special, urgent tasks from the hardware, and the execution is very brief. The relevance of NMI to containers is limited, therefore this study will not establish partition contexts for NMI execution contexts.

4.4 Establish Operation Partition Context

For different execution contexts, the method for establishing partition contexts and the location where they are stored differ. The location of the storage partition context needs to be ensured to be globally available to the execution context. This section describes the implementation of the partition context establishment in §3.2.

Process

Since process execution context has its own task context, partition context information can be stored in the task context. Since Linux namespaces are associated with processes and are available throughout the entire lifecycle of a process, the tracing namespace that already exists within the process's task context can serve as the partition context, eliminating the need to establish a separate partition context.

Kernel Thread

Since kernel thread execution context has its own task context, partition context information can be stored in the task context. Kernel threads do not originate directly from external operations, but are created by other kernel execution contexts. Therefore, kernel threads use the partition context inheritance in §3.3, which inherits the partition context from the execution context that created it.

Depending on the type of the parent execution context, copy the partition context from the correct location to its own task context (use the same method as in §4.5 to find the correct location). Not all kernel threads are created by a parent execution context that has a valid partition context. For example, kernel threads (workqueues) are used in the power subsystem, in which case these kernel threads will not inherit the partition context.

In the case of workqueues, a kernel thread can execute multiple workqueue tasks, each of which may come from a different parent execution context. Therefore, the parent partition context is saved in workqueue task data structure when the task is created, and the partition context in the task context of the current kernel thread is updated according to the partition context in workqueue task data structure when each task starts to be executed, and the partition context is cleared when the task ends.

Soft Interrupt

Since soft interrupt execution context do not have its own task context, this study stores the partition context in a per-CPU variable. Per-CPU variables are globally available in the soft interrupt execution context. Since soft interrupts cannot be preempted by another

soft interrupt in the Linux kernel, and the partition context in the per-CPU variable is only used by system events of the same CPU, there is no need to implement a reentrancy mechanism or add locks. One soft interrupt may handle multiple tasks in the task queue. Each task may belong to a different partition context. The partition context per-CPU variable is set when each task starts execution and cleared when each task ends.

In soft interrupt execution contexts, the partition context is not always set to the tracing namespace, but can also be set to other types of namespaces. For example, in a soft interrupt that processes network packets, the partition context can be set to the network namespace where the current network interface is located, and the corresponding tracing namespace will not be looked up until a system event is triggered and delivered. When establishing the partition context, this study collects only the minimum critical information and avoids any time-consuming operations (such as lookups) to prevent negative performance impacts on hot paths. System tracing is not always enabled. When system tracing is disabled, system events will not be triggered. Therefore, any time-consuming operations will only be executed when there is a real need to process system events.

For tasklets and timers implemented based on soft interrupts, similar to workqueues, they are not originated directly from external operations. Therefore, they will inherit the partition context of the parent execution context that created them. Similarly, not all tasklets and timers are created by a parent execution context that has a valid partition context. In such cases, these tasklets and timers will not inherit the partition context.

4.5 System Tracing Event Delivery

This study adds a tracing event router that delivers system events to the corresponding Ftrace instance when a system event is triggered, as described in §3.4. The tracing event router needs to determine which partition context to use based on the (current) execution context in which the system event was triggered. In process execution contexts and kernel thread execution contexts, the partition context in task context should be used. In soft interrupt execution contexts, the partition context in the per-CPU variable should be used. Since process execution context and kernel thread execution context can be preempted by soft interrupts, different partition contexts may be available at the same time. Therefore, it is necessary to determine which partition context is the correct partition context for the current execution context.

In the Linux kernel, the current execution context can be determined based on the per-CPU counter `preempt_count`. The `preempt_count` is used to represent the current CPU preemption state and execution context. This study uses the macros `in_task()` and `in_softirq()` based on preempt_count to determine the current execution context. Hard interrupts and NMIs can preempt soft interrupts in the Linux kernel, but hard interrupts and NMIs are not considered cases under `in_softirq()`. Although ksoftirqd actually executes soft interrupts in a kernel thread, it still performs the same setup as a normal soft interrupt. Therefore, `in_softirq()` still returns true in ksoftirqd, and no additional checks are required.

After obtaining the correct partition context, all Ftrace instances that have enabled the current system event will be checked to determine the partition where the current system event is located within the scope of the tracing namespace recorded in the Ftrace instance. In addition, the global Ftrace instance (associated with the global tracing namespace) is always delivered. Delivering a system event to an Ftrace instance does not mean that it will eventually appear in the tracing logs, because the filtering mechanisms in the

Ftrace instance are still working, which may discard system events that do not meet the configured filters.

All system events triggered in execution contexts without partition context information will only be delivered to the global system tracing instance and will only be visible to the global system tracing instance (outside the container), not inside the container. For example, hard interrupts, NMI, kernel threads that do not inherit partition context.

4.6 Tracing Cgroup Controller

Linux cgroups enable container creators to bind a set of limits or parameters to container processes [55]. In this implementation, this study adds tracing cgroup controller as the implementation for sensitive information hiding in §3.5. The tracing cgroup controller is used to configure whether container processes are allowed to view various types of sensitive information in system tracing logs and to bind these parameters to container processes. When creating a container, the container creator should configure the tracing cgroup controller to determine which types of sensitive information should be hidden and add the container process's PID to the corresponding cgroup.

The tracing cgroup controller currently supports configuration of whether to hide kernel object addresses, whether to hide the function offsets that trigger system events, and whether to hide buffer data in the kernel. All of these can be configured through virtual files in the cgroup pseudo-filesystem. In Linux kernel version 6.19.0 (the version on which this study is based), the addresses of kernel objects in the tracing logs have been hashed for security reasons and the raw memory addresses are not displayed. Therefore, in Linux kernel version 6.19.0, if a container creator decides to enable kernel object address hiding, it is only for the purpose of preventing kernel objects from being associated between system events (via hashed addresses), not for hiding the raw memory addresses.

The parameters in the tracing cgroup controller are bound to the container processes, not the Ftrace instance. The original system event data in the ring buffer of the Ftrace instance will not be changed. When a process reads the system tracing logs in a Ftrace instance, the Ftrace instance will decide whether to hide specific types of sensitive information in the returned system tracing logs based on the tracing cgroup controller parameters bound to the process. For the system administrator's processes (in the root cgroup), all sensitive information can be seen in the system tracing logs of the Ftrace instance. Sensitive information hiding only applies to processes within containers.

Chapter 5

Evaluation

This chapter describes the evaluation of TraceNS in this study. This study experimentally evaluates TraceNS in three aspects: performance, system event coverage, and security. These three aspects determine whether partitioned system tracing infrastructures such as TraceNS can be used in multi-tenant container environments and allow developers to gain insights from the behavior of containers.

This study used the Linux distribution Debian 13 and the Linux kernel (version 6.19.0) with the TraceNS prototype added. The experiments of this study were conducted on a computer with an i7-12700H CPU, 32GB of RAM, and a 512GB NVMe SSD.

5.1 Performance

Performance is critical for the system tracing infrastructure in operating systems. System tracing methods that introduce excessive overhead will distort the runtime behavior of containers, resulting in invalid measurements obtained through system tracing [34]. If the overhead of system tracing is too high, tracing high-frequency system events can even reduce the overall runtime performance of the system.

In addition to runtime performance, container creation performance is also important. In the microservice architecture, a web service can consist of a large number of containers running microservice components with different functions. The number of containers can be expanded or shrunk according to the current load. When some containers malfunction, new containers will be created to replace them. The performance of container creation determines how quickly the web service can adapt to burst loads and achieve self-healing.

This section describes the experimental performance evaluation of TraceNS in container creation and container runtime in this study. This study uses box plots to show the statistical distribution of performance data for each benchmark, representing minimum, first quartile, median, third quartile, and maximum in the plot. The median value for the corresponding data type is displayed in the x-axis labels of the box plot.

In some benchmarks, since the performance difference between the two scenarios is small, random factors in the system (such as scheduling) have a decisive impact on the final result, which can lead to some counterintuitive situations. For example, after enabling system tracing, the median performance is slightly higher than when it is not enabled. Although intuitively, system tracing should incur overhead and cause performance degradation, the counterintuitive situation exists statistically because the overhead is low.

5.1.1 Container Creation

The prototype implementation of TraceNS in this study introduces tracing namespace (§4.1), Ftrace instance creation (§4.2), Ftrace instance mounting (§4.2), and tracing cgroup controller (§4.6) during container creation.

This study compares the performance of the introduced creation step with other steps of the same type. For each comparison, 1,000 experiments are conducted for each scenario, with an additional 10 discarded warm-up runs.

Namespaces

This study compared the performance of tracing namespaces (§4.1) and other namespace types in the `unshare()` [69] and `clone3()` [56] system calls, as shown in Figure 5.1 and Figure 5.2.

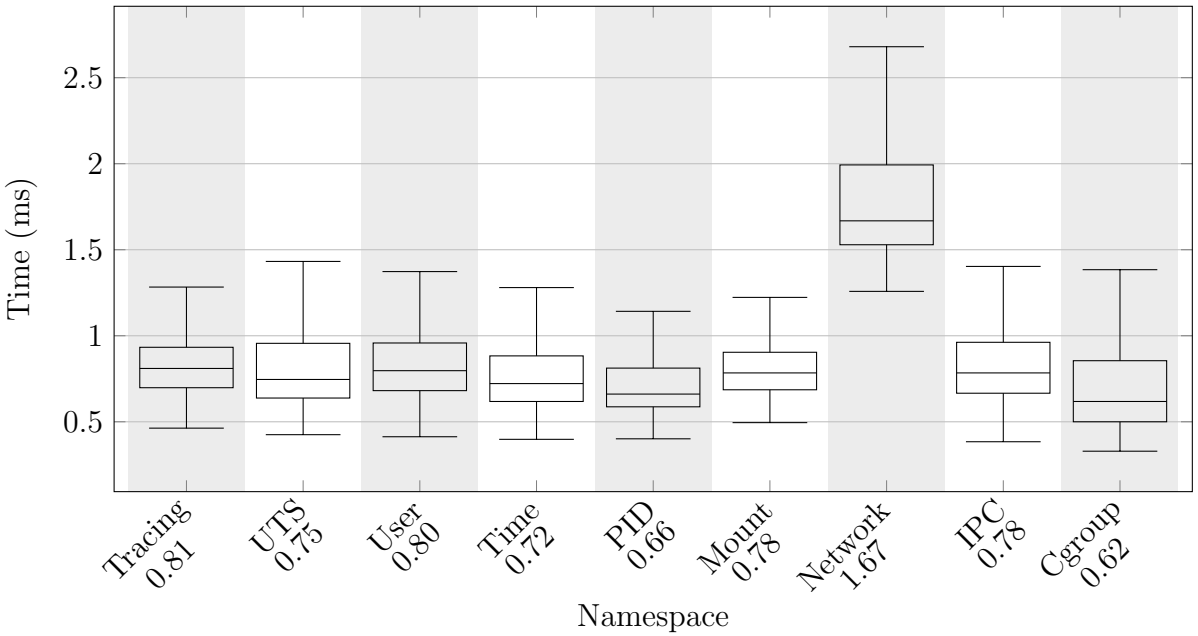

Figure 5.1: A box plot compares the creation overhead of the tracing namespace with that of other namespaces in the `clone3()` system call. The median is displayed in the x-axis labels in milliseconds (ms).

The `unshare()` system call creates a new namespace and moves the calling process into it. This new namespace will not be shared with any other process. This study measured the elapsed time from the start of the `unshare()` system call to its return.

The `clone3()` system call creates a new namespace and a child process. The parent process remains in its original namespace, while the child process moves to the new namespace. This study calls `clone3()` to create a child process that exits immediately and measures the elapsed time from the start of the `clone3()` system call to the exit of the child process.

As shown in Figure 5.1 and Figure 5.2, the experimental results indicate the network namespace have significant creation overhead (network namespace require the creation and initialization of extensive data structures, such as routing tables, firewall rule tables, etc.) and the creation overhead of tracing namespace is close to that of other namespace types. For the `unshare()` system call, the creation overhead of tracing namespace is slightly lower than that of IPC and Mount namespaces .

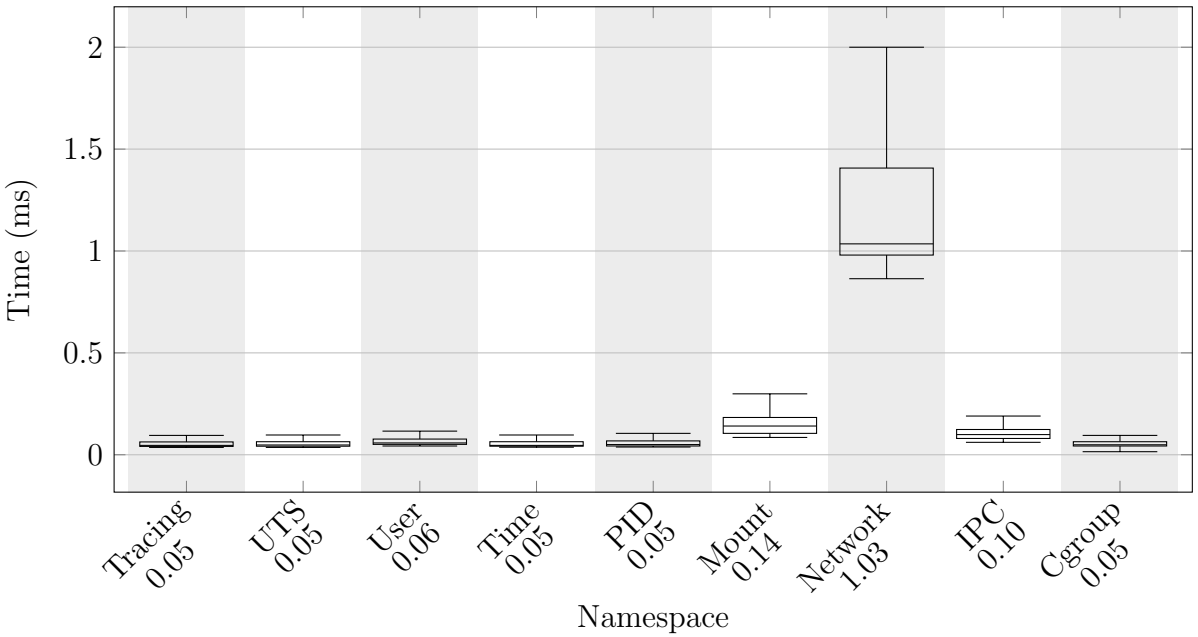

Figure 5.2: A box plot compares the creation overhead of the tracing namespace with that of other namespaces in the `unshare()` system call. The median is displayed in the x-axis labels in milliseconds (ms).

Ftrace Instance

During container creation, container creators typically need to mount procfs, devfs, sysfs, and tmpfs inside the container to ensure that applications that depend on these pseudo-filesystems can function correctly. In addition, in order for the data in the container to be preserved, the container creator usually needs to mount some external ordinary directories into the container as data directories. This study compared the performance of creating and mounting an Ftrace instance (§4.2) with mounting an external ordinary directory and common pseudo-filesystems, as shown in Figure 5.3.

For pseudo-filesystems, this study used the `mount()` system call [59] to mount and measured the elapsed time from the start of the `mount()` system call to its return.

For the ordinary directory, this study used a combination of the `open_tree()` [63] and `move_mount()` [60] system calls for mounting and measures the total elapsed time of the two system calls. The combination of the `open_tree()` and `move_mount()` system calls is a safe way provided by the Linux kernel to mount external directories into containers.

For the Ftrace instance, this study used the `mkdir()` system call [58] to create the Ftrace instance directory, then used a combination of `open_tree()` and `move_mount()` system calls to mount the Ftrace instance directory, and measures the total elapsed time.

As shown in Figure 5.3, the experimental results indicate the sysfs has noticeably higher overhead (sysfs contains a complex hierarchical device model mapping) and the overhead of creating and mounting an Ftrace instance with lazy initialization optimization is close to that of mounting other file systems. The overhead of creating and mounting an Ftrace instance only slightly higher than that of mounting an ordinary directory.

Cgroups

This study compared the performance of the tracing cgroup controller (§4.6) with other cgroup controllers during cgroup creation, as shown in Figure 5.4.

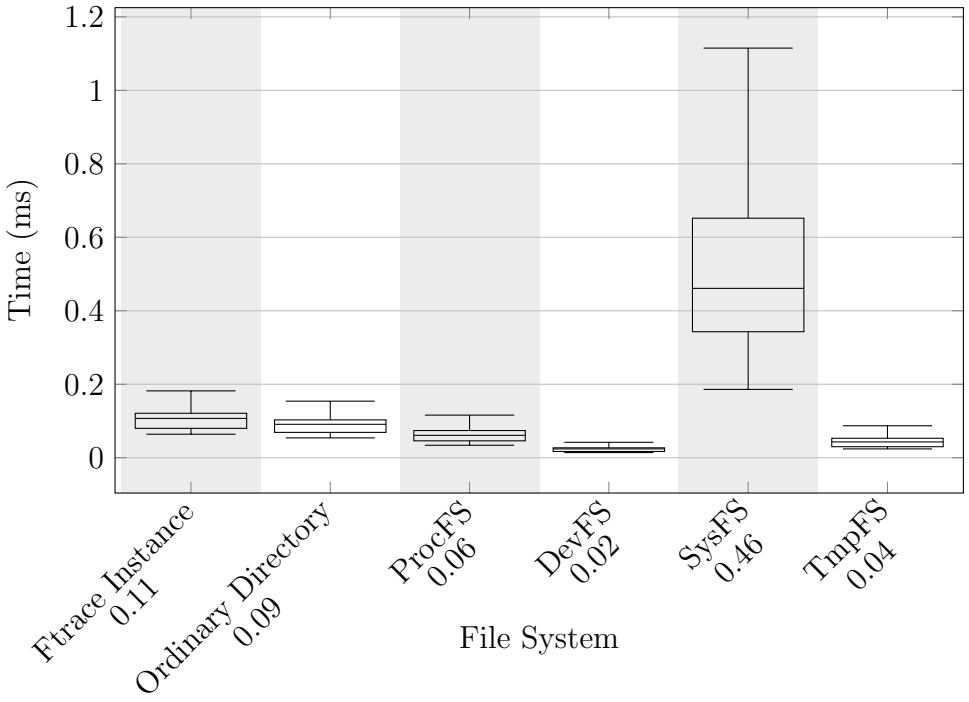

Figure 5.3: A box plot compares the overhead of creating and mounting an Ftrace instance with that of mounting other file systems or an external ordinary directory. The median is displayed in the x-axis labels in milliseconds (ms).

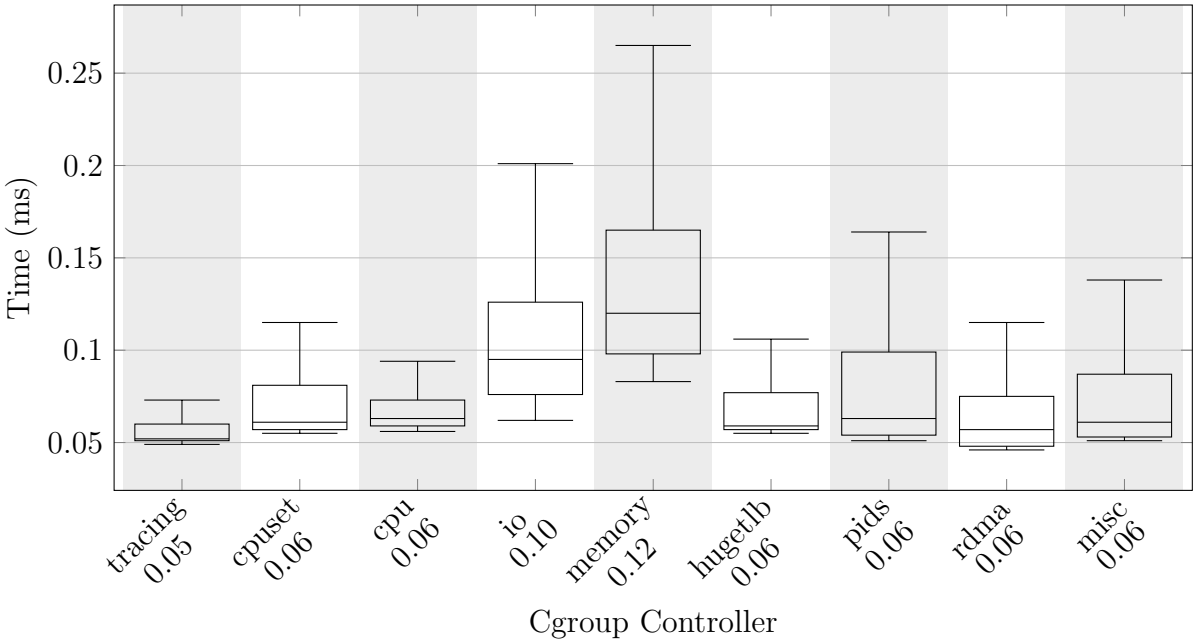

Figure 5.4: A box plot compares the creation overhead of the tracing cgroup controller with that of other cgroup controllers during cgroup creation. The median is displayed in the x-axis labels in milliseconds (ms).

This study controls the enabled cgroup controllers in child cgroups by modifying the `cgroup.subtree_control` file in the parent cgroup directory [55], creates child cgroup directories via the `mkdir()` system call [58], and measures the elapsed time from the start of the `mkdir()` system call to its return.

As shown in Figure 5.4, the experimental results indicate that the creation overhead of the tracing cgroup controller is relatively low among various cgroup controllers. This is because the tracing cgroup controller is only used to provide switches to hide sensitive information in the system tracing logs.

5.1.2 Container Runtime

This study evaluates the impact of TraceNS on container runtime performance by measuring network performance and file I/O performance within containers when system tracing is enabled. In the evaluation, this study creates 100 containers and compares eight different tracing scenarios:

- **O:** The benchmark program is running outside (**O**) the container. No Ftrace instance is tracing (outside-container baseline).
- **OG:** The benchmark program is running outside (**O**) the container. The global (**G**) Ftrace instance is tracing relevant system events.
- **C:** The benchmark program is running inside the container (**C**). No Ftrace instance is tracing (in-container baseline).
- **CG:** The benchmark program is running inside the container (**C**). The global (**G**) Ftrace instance is tracing relevant system events.
- **CC1:** The benchmark program is running inside the container (**C**). The Ftrace instance within one (**1**) container (**C**) is tracing relevant system events.
- **CC10:** The benchmark program is running inside the container (**C**). The Ftrace instances within ten (**10**) containers (**C**) are tracing relevant system events.
- **CC100:** The benchmark program is running inside the container (**C**). The Ftrace instances within all (**100**) containers (**C**) are tracing relevant system events.
- **CC1-99:** The benchmark program is running inside the container (**C**). The Ftrace instance within one (**1**) container (**C**) is tracing relevant system events, while the Ftrace instances within remaining containers (**99**) are tracing other system events.

For each comparison, the study conducted a 100-second experiment for each tracing scenario, recording the runtime performance per second (throughput), with an additional discarded 10-second warm-up.

Network Performance

This study used the iperf3 [81] tool to measure network performance and conducted experiments in three common scenarios of communication between containers:

- **Local Host:** Containers communicate with other containers on the same machine.
- **Data Center:** Containers communicate with other containers on other machines in the same data center.
- **Internet:** Containers communicate with other containers or users via the Internet.

The network environments for the three communication scenarios are:

- **Local Host:** No bandwidth limitations, no additional latency, no packet loss.
- **Data Center:** 50 Gbits bandwidth, 50 us latency (10 us jitter), 0.001% packet loss.
- **Internet (fibre):** 1000 Mbits bandwidth, 15 ms latency (2 ms jitter), 0.1% packet loss.

For benchmark programs running inside containers (**C**), the iperf3 client and server run in two separate containers, communicating with each other via veth [76] devices (virtual Ethernet device). For benchmark programs running outside (**O**) of containers, the iperf3 client and server communicate through the lo (loopback) interface.

For data center and Internet network environments, this study uses tc-netem (network emulator) [53] for simulation. Since the lo interface is a special interface with only one qdisc (queueing discipline), tc-netem behaves differently on the lo interface and the veth interface (the veth interface has two qdiscs at two endpoints). Therefore, in the simulated environments of data centers and the Internet, this study does not compare the tracing scenarios of benchmark programs running outside (**O**) the containers.

This study conducted experiments in each tracing scenario for each network environment, as shown in Figure 5.5, Figure 5.6, and Figure 5.7. The relevant tracing events enabled are `sock_recv_length` and `sock_send_length`. "Send" is the result obtained from the iperf3 client, while "Recv" is the result obtained from the iperf3 server.

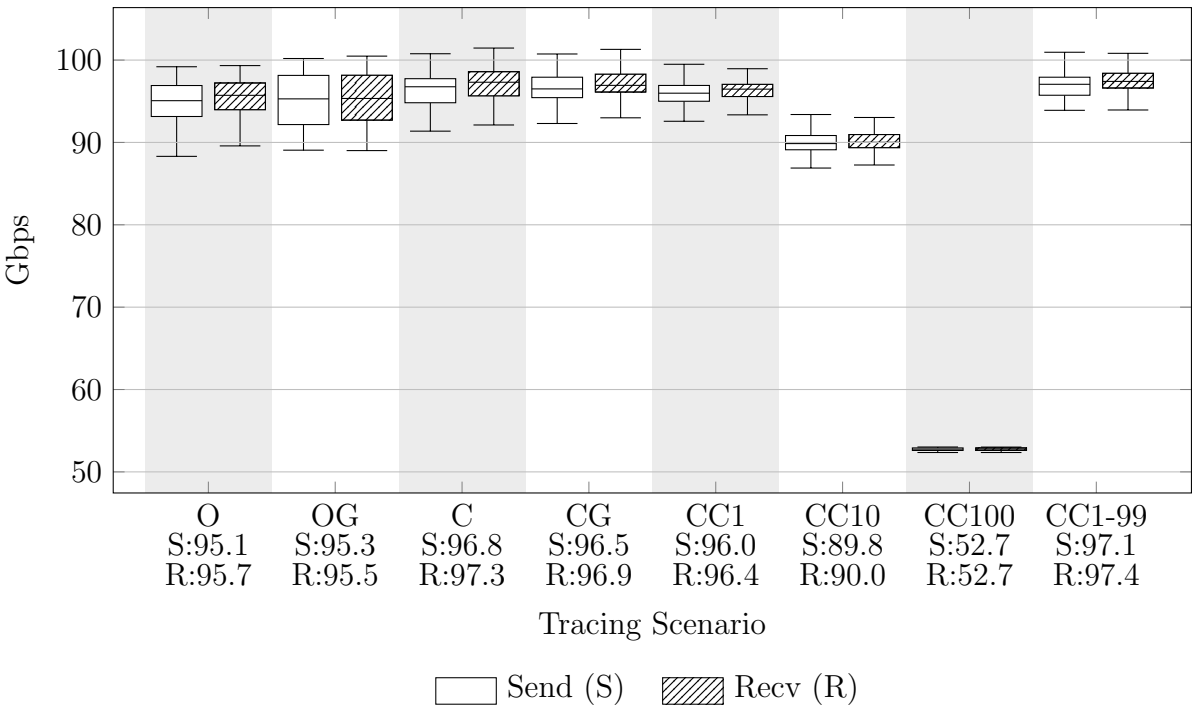

Figure 5.5: A box plot compares the network send/receive performance of the local host network in various tracing scenarios. The median network send (S) and receive (R) throughput are shown in the x-axis labels in Gbps. Since the tracing overhead is low, it is believed in this study that random factors (such as scheduling) in the system lead to some counterintuitive results. For example, the median send throughput outside the container is slightly higher when global tracing is enabled than when it is not.

As shown in Figure 5.5, the experimental results indicate that in the local host commu-

nication scenario, there is no noticeable difference in network performance between the benchmark program running outside the container (**O**) or inside the container (**C**), and with no tracing (**O**, **C**), global tracing (**OG**, **CG**), or one container tracing (**CC1**). Therefore, the additional overhead of containers and system operation partitions in TraceNS is shown to be low.

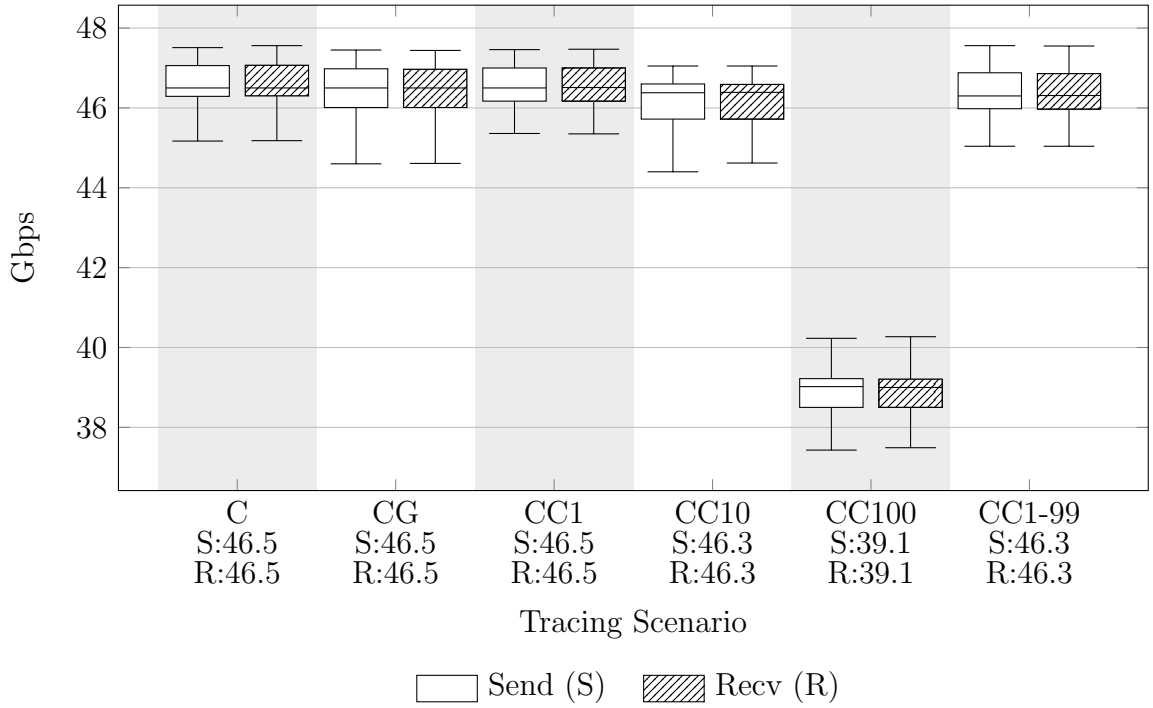

Figure 5.6: A box plot compares the network send/receive performance of the data center network in various tracing scenarios. The median network send (S) and receive (R) throughput are shown in the x-axis labels in Gbps.

When 10 containers are tracing (**CC10**), network performance showed a slight decrease. When 100 containers are tracing (**CC100**), a significant performance drop occurred. This is because delivering events to the buffers of 100 Ftrace instances inevitably incurs overhead. Since this is a local host communication scenario, all data packet processing is done solely in the CPU, and network packets are not processed by real network devices. Therefore, CPU overhead is a significant factor affecting network performance.

When only one container is tracing relevant events while other containers are tracing irrelevant events, there is no noticeable performance degradation. This is because when an event is triggered, only the Ftrace instances that are tracing that event are traversed, and Ftrace instances that are tracing other events do not incur any overhead.

As shown in Figure 5.6, the experimental results indicate that in data center communication scenario, when 10 containers are tracing (**CC10**), the decline in network performance is smaller compared to that of the local host. Only when 100 containers are tracing (**CC100**) does a noticeable decline in network performance. This is because the maximum network bandwidth in data center communication scenarios is lower than that of the local host.

As shown in Figure 5.7, the experimental results indicate that in Internet communication scenarios, even if 100 containers are tracing (**CC100**), there is no noticeable impact on network performance. This is because the main performance bottleneck in the Internet

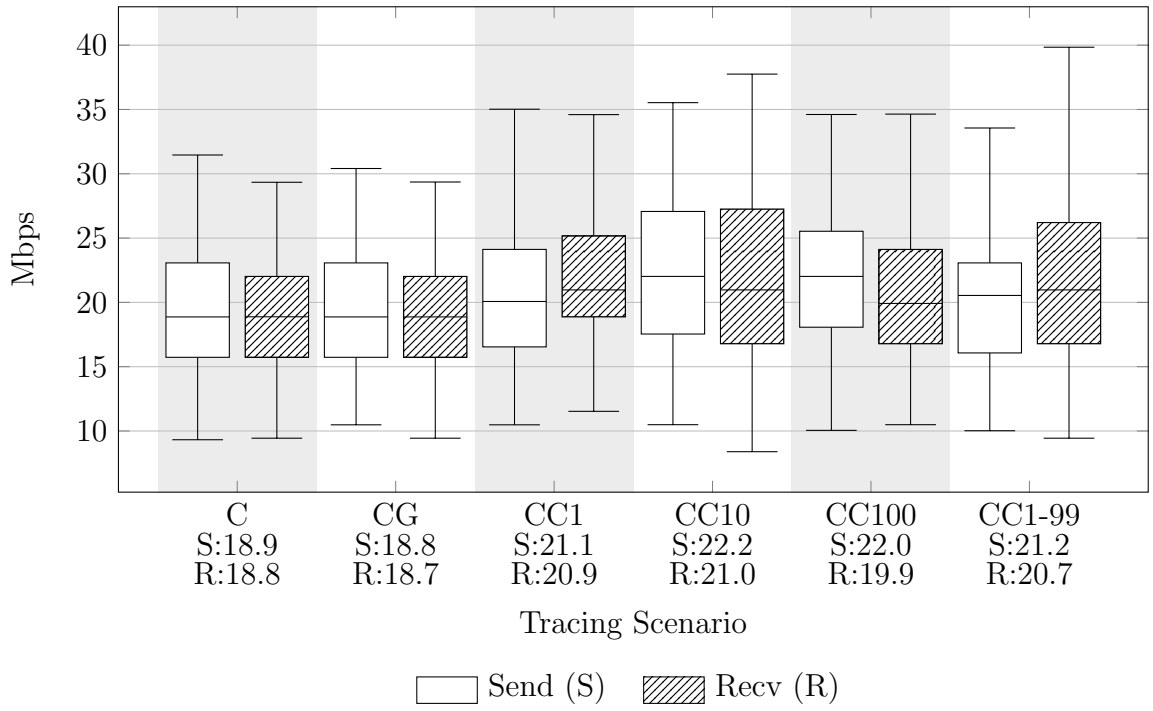

Figure 5.7: A box plot compares the network send/receive performance of the internet in various tracing scenarios. The median network send (S) and receive (R) throughput are shown in the x-axis labels in Mbps.

communication scenario is network bandwidth, and random network jitter and packet loss can also have a significant impact on network performance.

File I/O Performance

This study used the fio [4] tool to measure file read/write performance, with psync as the I/O engine, 5GB file size, 1 MB block size, and unbuffered I/O. This study measured sequential read/write performance and random read/write performance in containers.

This study conducted experiments for each tracing scenario of each read/write mode, as shown in Figure 5.8 and Figure 5.9. For reads, the relevant tracing events enabled are `sys_enter_pread64` and `sys_exit_pread64`. For writes, the relevant trace events enabled are `sys_enter_pwrite64` and `sys_exit_pwrite64`.

As shown in Figure 5.8 and Figure 5.9, the experimental results indicate that even if all 100 containers are tracing (**CC100**) relevant system events, there is still no noticeable impact on file I/O performance. This is because although the tracing overhead still exists, the performance bottleneck is disk bandwidth rather than CPU.

5.2 System Event Coverage

For system tracing infrastructure in container environments, only by covering enough container-related system events can it enhance system-level observability in the container environment, help developers understand container-related system behavior, and help developers analyze, debug, and optimize containers.

This section describes the experimental system event coverage evaluation of TraceNS for system calls and predefined system events in kernel subsystems. In the Linux kernel, dif-

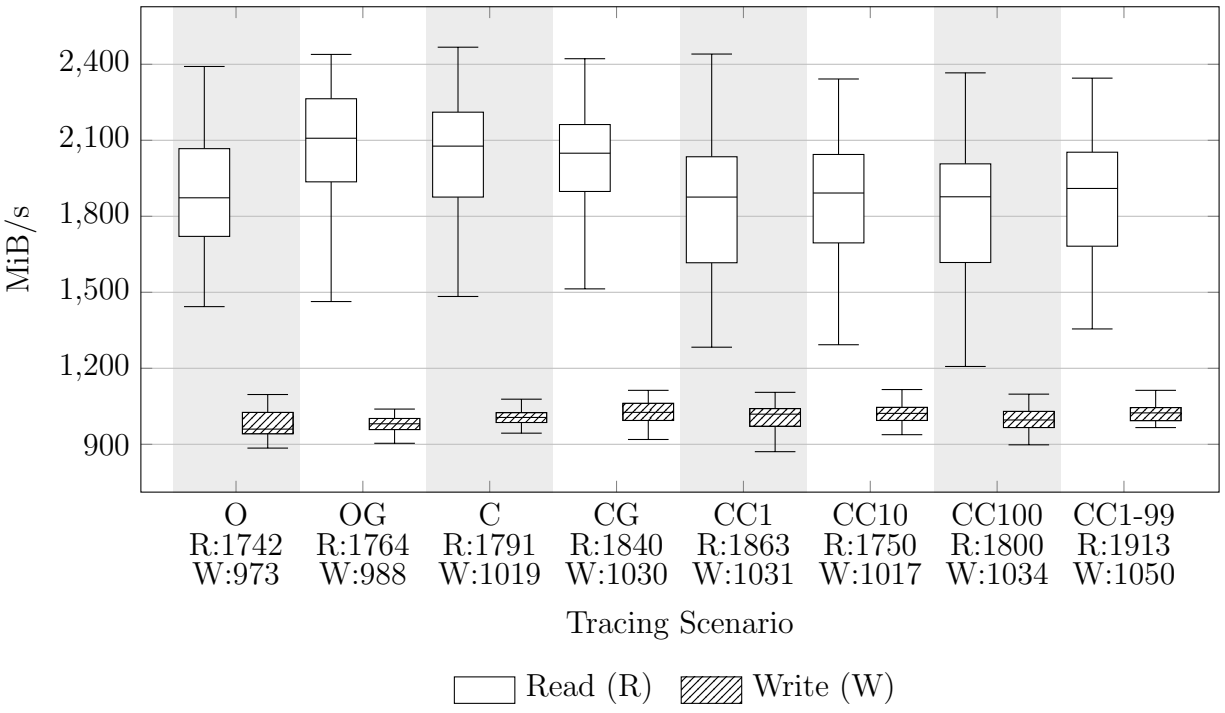

Figure 5.8: A box plot compares the sequential read/write performance of disk files under various tracing scenarios. The median file read (R) and write (W) throughput are shown in the x-axis labels in MiB/s.

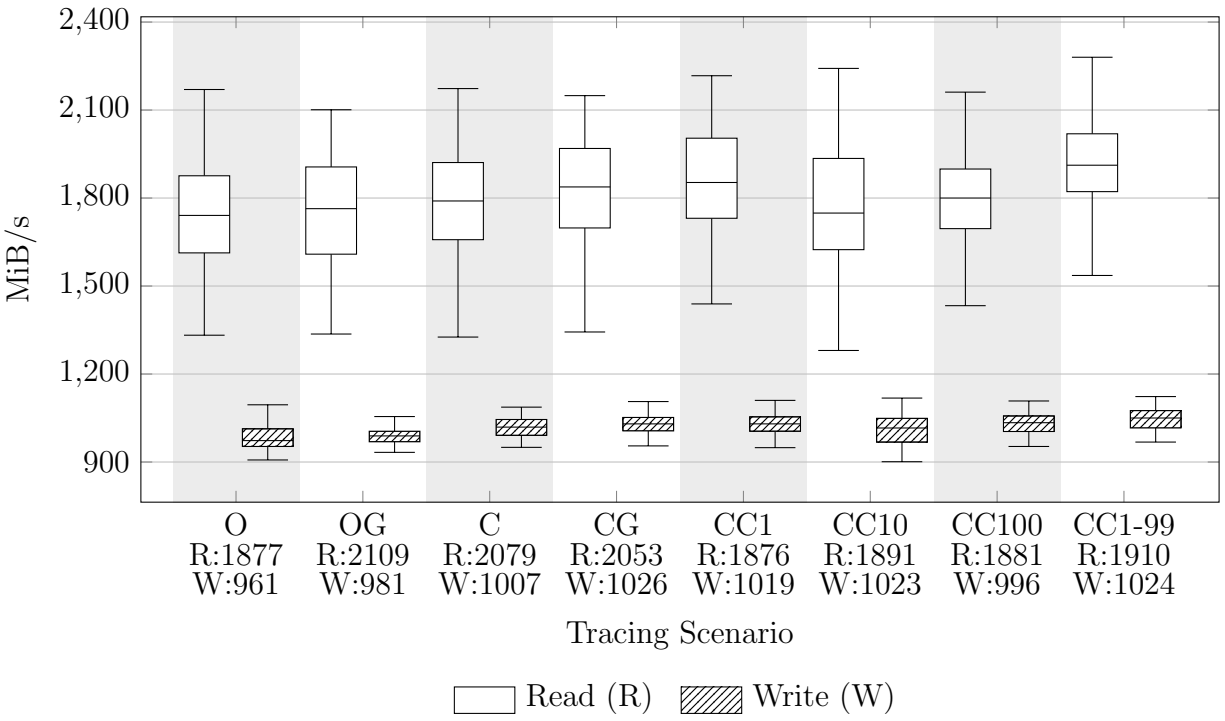

Figure 5.9: A box plot compares the random read/write performance of disk files under various tracing scenarios. The median file read (R) and write (W) throughput are shown in the x-axis labels in MiB/s.

ferent kernel compilation configurations and different architectures can affect the number of system calls and predefined system events available in kernel subsystems. This evaluation is based on the kernel compilation configuration of the Linux distribution Debian 13 on the x86_64 (amd64) architecture.

5.2.1 System Call

The method used in this study to evaluate system call event coverage is to enable the events for a system call within a container, call that system call in the container, and then check the tracing log in the container to find whether the system call that was just called exists in the log. If it exists, it means that this system call can be traced within the container, and TraceNS has covered this system call. Table 5.1 shows the experimental results of system call event coverage.

All system call events can be found in `/sys/kernel/tracing/events/syscalls/`. Each system call has `sys_enter` and `sys_exit` events, representing the entry and exit of a system call, respectively. This study checks each `sys_enter` and `sys_exit` event separately.

This study sets all parameters to `0xFFFFFFFF` when each system call is called. Since this is an event coverage evaluation, which only checks whether the enter and exit events of system calls can be traced, the calling method similar to fuzzing [103] can be used.

Event Type	Testable Events	Covered Events
sys_enter_*	355/356	355/355
sys_exit_*	355/356	352/355

Table 5.1: System call coverage in TraceNS. Testable events represent system calls that can be successfully triggered in the experimental environment. Covered events represent system calls that can be successfully partitioned by TraceNS. In the testable events column (`a/b`), `a` is the number of testable system calls, and `b` is the total number of system calls. In the covered events column (`a/b`), `a` is the number of covered system calls, and `b` is the number of testable system calls.

The Linux kernel version 6.19.0 compiled under the Debian 13 Linux distribution on the x86_64 architecture contains 356 system calls. The `vhangup()` [77] system call cannot be tested because it causes the current terminal to hangup and the evaluation program to malfunction. Therefore, only 355 system calls are testable.

As shown in Table 5.1, the experimental results indicate that the `sys_enter` event of all testable system calls has been covered. The `sys_exit` event of 352 system calls has been covered, and the remaining 3 system calls are `rt_sigreturn()` [74], `exit()` [70], and `exit_group()` [71]. These three system calls do not return, so even if the `sys_exit` event for these system calls exists, it will never be triggered.

5.2.2 Predefined System Events in Kernel Subsystems

The method used in this study to evaluate the coverage of predefined system events in kernel subsystems is to write a workload program in user space that can trigger system events in a kernel subsystem, enabling the events of this kernel subsystem in a container, running the workload program in the container, and then checking the tracing logs in the container to find whether the corresponding system events exists in the log. A kernel

subsystem may have multiple system events, so this study checks each system event separately. If a system event exists, it means that this system event can be traced within the container, and TraceNS has covered this system event.

Although loading kernel modules can more easily trigger system events, this can cause system events to be triggered in a abnormal context, resulting in invalid coverage. Therefore, this study chose to trigger system events through userspace programs, which ensures that system events are triggered in a normal context.

The Linux kernel version 6.19.0 compiled under the Debian 13 Linux distribution on the x86_64 architecture contains predefined system events in 140 kernel subsystems. All predefined system events in kernel subsystems can be found in `/sys/kernel/tracing/events/`. System call coverage has already been evaluated in §5.2.1. Excluding the `syscalls` and `raw_syscalls` subsystems, there are 138 kernel subsystems remaining.

Of these 138 kernel subsystems, many have limited relevance to containers. For example, cpuhp (CPU hot-plugging), initcall (kernel initialization call), regulator (voltage regulation). System events in these kernel subsystems are unlikely to be triggered by container behavior and unlikely to be partitioned, therefore this study did not evaluate system event coverage for these kernel subsystems.

This study found that 39 of the 138 kernel subsystems are related to containers, and system events in these subsystems can be triggered by container behavior. Due to time constraints, this study evaluated the coverage of predefined system events in 23 container-related kernel subsystems, and Table 5.2 shows the evaluation results.

Some predefined system events in kernel subsystems are used to report errors or can only be triggered in special hardware. For example, `net_dev_xmit_timeout` in the net subsystem indicates that the network interface cannot transmit data packets within a reasonable time, which normally does not occur. Therefore, these events are difficult to trigger in experimental environments of this study and are left for future work.

As shown in Table 5.2, the experimental results indicate that TraceNS has covered all testable system events in the evaluated container-related kernel subsystems.

Subsystem	**Testable Events**	**Covered Events**
task	3/3	3/3
signal	2/2	2/2
workqueue	4/4	4/4
lock	2/2	2/2
timer	12/14	12/12
filemap	5/7	5/5
mmap_lock	3/3	3/3
mmap	2/2	2/2
filelock	12/12	12/12
fuse	2/2	2/2
io_uring	17/17	17/17
neigh	6/7	6/6
bridge	5/5	5/5
qdisc	5/5	5/5

Subsystem	Testable Events	Covered Events
fib	1/1	1/1
udp	1/1	1/1
sock	5/7	5/5
net	13/16	13/13
skb	3/3	3/3
netlink	1/1	1/1
icmp	1/1	1/1
fib6	1/1	1/1
mptcp	5/5	5/5

Table 5.2: Predefined system event coverage in TraceNS. Testable events represent predefined system events that can be successfully triggered in the experimental environment. Covered events represent predefined system events that can be successfully partitioned by TraceNS. In the testable events column (`a/b`), `a` is the number of testable predefined system events in a subsystem, and `b` is the total number of predefined system events in a subsystem. In the covered events column (`a/b`), `a` is the number of covered predefined system events in a subsystem, and `b` is the number of testable predefined system events in a subsystem.

5.3 Security

In multi-tenant container environments, the security of the system tracing infrastructure is critical. The lack of true isolation of system tracing event views prevents the system tracing infrastructure in operating systems from being available within containers. System tracing events may contain sensitive information, which needs to be hidden to prevent potential exploitation by untrusted container tenants.

This section describes the experimental security evaluation of TraceNS for system tracing event view isolation and sensitive information hiding in this study. The security evaluation in this section is based on the covered system events in §5.2.

5.3.1 Event Isolation

This study evaluated the event isolation of TraceNS in three aspects:

- **Isolated (Global):** Global system events that are outside the container cannot be seen inside the container.
- **Isolated (Container):** System events inside one container cannot be seen from another container.
- **Visible (Global):** System events inside the container can be seen from outside the container.

The method used in this study to evaluate event isolation is to create container A and container B, enable relevant system events outside the containers (global), in container A, and in container B, run the same workload as in §5.2 outside the containers and inside container A, and check the system tracing logs outside the containers (global), in container A, and in container B.

After the workload runs outside the containers, if there are relevant system events in the system tracing logs outside the containers, but not in the system tracing logs in container A and container B, then `Isolated (Global)` is achieved.

After the workload runs in container A, if there are relevant system events in the system tracing logs outside the containers and in container A, but not in the system tracing logs in container B, then `Isolated (Container)` and `Visible (Global)` are achieved.

This study evaluated the event isolation of TraceNS for system call events and predefined system events in kernel subsystems, as shown in Table 5.3 and Table 5.4, respectively.

As shown in Table 5.3 and Table 5.4, the experimental results indicate that TraceNS has achieved three aspects of event isolation for all covered system call events and predefined system events in kernel subsystems.

Event Type	**Isolated (Global)**	**Isolated (Container)**	**Visible (Global)**
sys_enter_*	355/355	355/355	355/355
sys_exit_*	352/352	352/352	352/352

Table 5.3: The isolation of system call events in TraceNS. In all columns (`a/b`), `b` is the number of system calls covered by TraceNS, and `a` is the number of system calls that meet the TraceNS event isolation requirement for the current column.

Subsystem	**Isolated (Global)**	**Isolated (Container)**	**Visible (Global)**
task	3/3	3/3	3/3
signal	2/2	2/2	2/2
workqueue	4/4	4/4	4/4
lock	2/2	2/2	2/2
timer	12/12	12/12	12/12
filemap	5/5	5/5	5/5
mmap_lock	3/3	3/3	3/3
mmap	2/2	2/2	2/2
filelock	12/12	12/12	12/12
fuse	2/2	2/2	2/2
io_uring	17/17	17/17	17/17
neigh	6/6	6/6	6/6
bridge	5/5	5/5	5/5
qdisc	5/5	5/5	5/5
fib	1/1	1/1	1/1
udp	1/1	1/1	1/1
sock	5/5	5/5	5/5
net	13/13	13/13	13/13
skb	3/3	3/3	3/3
netlink	1/1	1/1	1/1
icmp	1/1	1/1	1/1

Subsystem	Isolated (Global)	Isolated (Container)	Visible (Global)
fib6	1/1	1/1	1/1
mptcp	5/5	5/5	5/5

Table 5.4: The isolation of predefined system events in TraceNS. In all columns (`a/b`), `b` is the number of predefined system events in a subsystem covered by TraceNS, and `a` is the number of predefined system events in a subsystem that meet the TraceNS event isolation requirement for the current column.

5.3.2 Sensitive Information Hiding

This study evaluated the sensitive information hiding of TraceNS in two aspects:

- **Desensitized (Container):** Sensitive information that is not allowed to be seen within a container should be hidden.
- **Not Desensitized (Global):** All sensitive information should be visible outside the container.

The method used in this study to evaluate sensitive information hiding is to create a container, enable all sensitive information hiding for this container using the tracing cgroup controller, enable relevant system events outside the container (global) and inside the container, run the same workload as in §5.2 inside the container, and check the system tracing logs outside the container (global) and inside the container.

After the workload runs inside the container, if the system tracing logs outside the container contain sensitive information, then `Not Desensitized (Global)` is achieved. If all sensitive information is hidden in the system tracing logs inside the container, then `Desensitized (Container)` is achieved.

Since all system calls originate from userspace programs, the object addresses and buffer data in the system call parameters, and the functions that called the system calls, all come from userspace programs and do not need to be hidden from userspace programs. In the TraceNS prototype of this study, sensitive information hiding for system calls was neither implemented nor evaluated.

Subsystem	Desensitized (Container)	Not Desensitized (Global)
task	N/A	N/A
signal	N/A	N/A
workqueue	4/4	4/4
lock	2/2	2/2
timer	10/10	10/10
filemap	N/A	N/A
mmap_lock	3/3	3/3
mmap	1/1	1/1
filelock	12/12	12/12
fuse	N/A	N/A
io_uring	17/17	17/17
neigh	N/A	N/A

Subsystem	Desensitized (Container)	Not Desensitized (Global)
bridge	N/A	N/A
qdisc	2/2	2/2
fib	N/A	N/A
udp	N/A	N/A
sock	2/2	2/2
net	9/9	9/9
skb	3/3	3/3
netlink	N/A	N/A
icmp	1/1	1/1
fib6	N/A	N/A
mptcp	1/1	1/1

Table 5.5: The sensitive information hiding of predefined system events in TraceNS . In all columns (`a/b`), `b` is the number of predefined system events in a subsystem covered by TraceNS that contain sensitive information, and `a` is the number of predefined system events that meet the TraceNS sensitive information hiding requirement for the current column. Kernel subsystems that do not contain system events containing sensitive information are indicated as `N/A`.

This study only evaluated the sensitive information hiding for predefined system events in kernel subsystems, as shown in Table 5.5. Not all system events contain sensitive information. This study only evaluated system events that did contain sensitive information. Not all kernel subsystems have system events containing sensitive information. Kernel subsystems that do not contain system events containing sensitive information are indicated in the table as `N/A`.

As shown in Table 5.5, the experimental results indicate that TraceNS has achieved two aspects of sensitive information hiding for all covered predefined system events in kernel subsystems.

5.4 Summary

This study evaluated TraceNS in terms of container creation performance (§5.1.1), container runtime performance (§5.1.2), system call coverage (§5.2.1), predefined system event coverage (§5.2.2), event isolation (§5.3.1), and sensitive information hiding (§5.3.2).

During container creation, TraceNS has similar or relatively lower overhead compared to other steps of the same type (namespaces, mounts, cgroups). During container runtime, TraceNS has low overhead in most scenarios, and only has notable overhead when 100 containers are tracing the same event. It is assumed in this study that it is unlikely in real-world scenarios that many containers from different tenants are tracing the same event at the same time. In real-world scenarios, it is more likely that containers from different tenants are tracing different events at the same time.

For system event coverage, TraceNS covered all testable system calls and all testable predefined system events in evaluated kernel subsystems. This demonstrates that the approach of the system operation partition context in TraceNS is scalable.

For security, TraceNS achieved event isolation and sensitive information hiding in all covered system events. This demonstrates that TraceNS can be used securely in multi-tenant container environments without causing cross-container privacy information leaks or disclosing sensitive information that could potentially compromise system security.

Chapter 6

Case Studies

This chapter describes case studies of TraceNS, demonstrating how TraceNS can help developers analyze, debug, and optimize containers in real-world scenarios. All the case studies provided in this study replicate containers in real-world production scenarios, where it is difficult to diagnose containers and find the cause of the issues when no system tracing infrastructure is available.

All tracing logs shown in this chapter are from the extended Ftrace in the TraceNS prototype implementation of this study. Since tracing logs are typically long, this chapter only shows the parts most relevant to case studies, omitting the rest. Some case studies use the function graph tracer [23] in Ftrace, which displays function call relationships similar to C source code. The left curly brace indicates the start of function execution, and the right curly brace indicates the end of function execution. Functions called within a function are nested within curly braces.

6.1 Eliminating Container Long-tail Latency

This case study describes the long-tail latency issue of an online gaming company. In this online gaming company, all servers are deployed on a cloud platform.

This online gaming company deployed a log server, which stores logs from all other game backend servers. Since the game server generates a large number of logs, the log server writes a large amount of data to the disk every second. After each successful log write, the log server replies with an "OK" message to the sender.

Engineers noticed that the log server sometimes took a long time to reply with an "OK" message, sometimes 1000 ms, and sometimes even 2000 ms. However, in most cases, the log server responds quickly, taking only 2 ms - 5 ms. This is a long-tail latency issue that engineers need to eliminate.

Engineers reviewed the source code of the log server but found no problems. Engineers believed this is a system-level issue and attempted to resolve it through system tracing within the container.

Engineers first traced all system calls called by the log server. In the tracing logs, engineers discovered some `write()` [78] system calls that took more than 2000000 us (2000 ms) to complete, as shown in Figure 6.1. The kernel function starting with "ext4" indicates that the `write()` system call is writing to disk, which narrows down the issue.

```
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 ...
 7) $ 2543244 us  |          } /* ext4_buffered_write_iter */
 7) $ 2543247 us  |        } /* ext4_file_write_iter */
 7) $ 2543250 us  |      } /* vfs_write */
 7) $ 2543252 us  |    } /* ksys_write */
 7) $ 2543253 us  |  } /* __x64_sys_write */
 ...
```

Figure 6.1: Slow `write()` system calls are found in the Ftrace logs within the container.

Engineers performed another system tracing, this time focusing only on the kernel function `ext4_buffered_write_iter()`. Engineers discovered that these time-consuming kernel functions ultimately called `try_to_writeback_inodes_sb()`, as shown in Figure 6.2.

```
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 ...
 2) $ 2127642 us  |              } /* wb_wait_for_completion */
 2) $ 2127671 us  |            } /* __writeback_inodes_sb_nr */
 2) $ 2127677 us  |          } /* try_to_writeback_inodes_sb */
 2) $ 2127679 us  |        } /* ext4_nonda_switch */
 2) $ 2127750 us  |      } /* ext4_da_write_begin */
 2) $ 2127775 us  |    } /* generic_perform_write */
 2) $ 2127779 us  |  } /* ext4_buffered_write_iter */
 ...
```

Figure 6.2: The Ftrace logs in the container revealed that the time-consuming `write()` system call ultimately called the kernel function `try_to_writeback_inodes_sb()`.

`try_to_writeback_inodes_sb()` is a dirty page write-back function in the Linux kernel. Since the log server continuously writes a large amount of log data to the disk, a large number of dirty pages are generated. When the number of dirty pages exceeds the threshold, the Linux kernel writes a large number of dirty pages to the disk at once, causing the long-tail latency issue.

After the engineers understood the cause, they modified the log server program to periodically and proactively inform the kernel that dirty pages were no longer needed (`posix_fadvise(POSIX_FADV_DONTNEED)`) [73], thus preventing the accumulation of dirty pages and successfully eliminating this long-tail latency issue.

6.2 Container Network Troubleshooting

This case study describes the transaction request failure issue of an e-commerce platform company. In this e-commerce platform company, the customer-facing shopping platform servers and the internal transaction servers are deployed in different data centers. Customers can only access the shopping platform server. When customers initiate transactions, shopping platform servers forward the transactions request to transaction servers.

Engineers noticed that during promotional events, when transaction volume increases dramatically in a short period of time, the TCP connections initiated by the shopping

platform server to the transaction server would sometimes time out, resulting in transaction failures, which affected the user experience.

Engineers reviewed the source code of the transaction server and shopping platform server but found no problems. Engineers also checked the network dashboard during the event but found no network congestion. Engineers believed this is a system-level issue and attempted to resolve it through system tracing within the container.

Engineers first traced the `kfree_skb` event to check whether TCP packets were being dropped in the kernel when TCP connections could not be established. In the tracing logs, engineers discovered that when transaction volume increased, many TCP packets were dropped due to `NEFILTER_DROP`, as shown in Figure 6.3.

```
# ... TIMESTAMP  FUNCTION
#         |          |
...
... 81.132304: kfree_skb: ... reason: NETFILTER_DROP
... 82.169985: kfree_skb: ... reason: NETFILTER_DROP
... 83.369769: kfree_skb: ... reason: NETFILTER_DROP
...
```

Figure 6.3: The cause of the TCP packets being dropped was found to be `NEFILTER_DROP` in the Ftrace logs within the container.

However, the engineers did not configure incorrect iptables rules, and it is unlikely that the cloud provider configured incorrect iptables rules either.

Engineers attempted to print the stack of `kfree_skb` event, as shown in Figure 6.4, but since it was kernel function `nf_hook_slow()` that dropped the packets, it did not provide much information.

```
 ...
 => trace_event_raw_event_kfree_skb
 => trace_kfree_skb
 => sk_skb_reason_drop
 => nf_hook_slow
 ...
```

Figure 6.4: The stack trace in the Ftrace logs within the container showed that the kernel function that drops TCP packets is `nf_hook_slow()` kernel function.

Engineers attempted to trace all netfilter-related kernel functions (`nf_*`) and eventually found the cause in the tracing logs, as shown in Figure 6.5. According to the tracing logs, packets were marked as to be dropped in `__nf_conntrack_alloc()`.

By checking the Linux kernel source code, it can be known that packets are marked for dropping in `nf_conntrack_in()` because the conntrack table is full. Since this is a cloud platform shared by many tenants and provides network access via NAT (Network Address Translation), conntrack table being full is a potential occurrence during peak periods.

After the engineers understood the cause, they modified the shopping platform server and transaction server, no longer creating a new TCP connection for each transaction, but instead reusing a small number of TCP connections to process all transactions, thereby successfully eliminating this issue.

```
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 ...
 6)               |  nf_hook_slow() {
 6)               |    ...
 6)               |    nft_do_chain_ipv4() {
 6)               |    ...
 6)               |    ipv4_conntrack_in() {
 6)               |    ...
 6)               |      nf_conntrack_in() {
 6)               |    ...
 6)               |          __nf_conntrack_alloc() {
 6)               |    ...
 6)               |            early_drop() {
 6)               |    ...
 6)               |    sk_skb_reason_drop() {
 ...
```

Figure 6.5: The Ftrace logs in the container revealed that TCP packets were marked as to be dropped in the `nf_conntrack_in()` kernel function.

6.3 Solving a Container Quirk

This case study describes the container first request quirk at a data science company. In this data science company, a data file and a statistics server program are packaged into a container and deployed on a cloud platform.

The statistics server provides an interface to receive new data, append the new data to the data file, perform calculations with the existing data, and return the latest statistical information. The data file is periodically downloaded and backed up from the statistics server, and then packaged into a container and deployed to the cloud platform the next time the engineers update the statistics server program.

Engineers noticed a long-standing quirk of the statistics server: every time the server program was updated and packaged into a container and deployed to the cloud platform, the first request would experience a long latency, lasting about 5 seconds, but this latency would not occur in subsequent requests.

Engineers reviewed the source code of the statistics server found no problems. Engineers believed this is a system-level issue and attempted to resolve it through system tracing within the container.

```
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 ...
 4) $ 5411232 us  |  } /* __x64_sys_open */
 ...
```

Figure 6.6: A slow `open()` system call is found in the Ftrace logs within the container.

Engineers first traced all system calls called by the statistics server during the first request. Engineers discovered an `open()` [72] system call in the tracing logs that took nearly 5 seconds to complete, as shown in Figure 6.6. This is a strange symptom. The `open()` system call only opens a file, not writes data, and usually does not take that long.

```
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 ...
 0) $ 5325847 us  |                    } /* ovl_copy_up_flags */
 0) $ 5325848 us  |                  } /* ovl_maybe_copy_up */
 0) $ 5325867 us  |                } /* ovl_open */
 0) $ 5325874 us  |              } /* do_dentry_open */
 0) $ 5325875 us  |            } /* vfs_open */
 0) $ 5326319 us  |          } /* path_openat */
 0) $ 5326320 us  |        } /* do_filp_open */
 0) $ 5326330 us  |      } /* do_sys_openat2 */
 0) $ 5326330 us  |    } /* do_sys_open */
 0) $ 5326331 us  |  } /* __x64_sys_open */
 ...
```

Figure 6.7: The Ftrace logs in the container revealed that the time-consuming `open()` system call ultimately called the kernel function `ovl_copy_up_flags()`.

Engineers attempted to trace `open()` system call and discovered that it internally calls `ovl_open()`, which ultimately calls `ovl_copy_up_flags()`, as shown in Figure 6.7.

`ovl_copy_up_flags()` is a kernel function in overlayfs [26]. In modern containers, the underlying container image is immutable. With overlayfs, all changes only affect the upper layers. If a file in the container image is modified, overlayfs needs to perform copy-on-write (CoW), constructing a new file internally.

Since the engineers mistakenly packaged a big data file into the container image and opens the data file with `O_RDWR` during the first request to the statistics server, overlayfs performed a time-consuming copy-on-write operation during the first request, making the first request take nearly 5 seconds.

After the engineers understood the cause, they stopped packaging data files into containers and instead stored them in an external data directory, which was then mounted into the container, successfully eliminating this container quirk.

6.4 Improving Container Performance

This case study describes the container network performance issue at a media company. Instead of hiring software engineers, this media company purchased a custom cloud storage service program using the TCP protocol from a software development company and deployed it on a cloud platform.

The media company's employees discovered that although the container service they purchased was supposed to support 100 Mbps (12.5 MB/s) of bandwidth, the actual download speed from the server was only about 1 MB/s. Since the media company lacked software engineers, it requested assistance from cloud platform engineers to analyze the problem.

Engineers checked the network and found no problems. Engineers did not receive any reports of network performance issues from other tenants on the same cloud server. Therefore, the engineers believed the problem lay within the container.

Since the engineers did not have the source code for the cloud storage service program, they attempted to perform system tracing for the container to understand its behavior.

Since the engineers had administrator privileges on the cloud servers, they first attempted

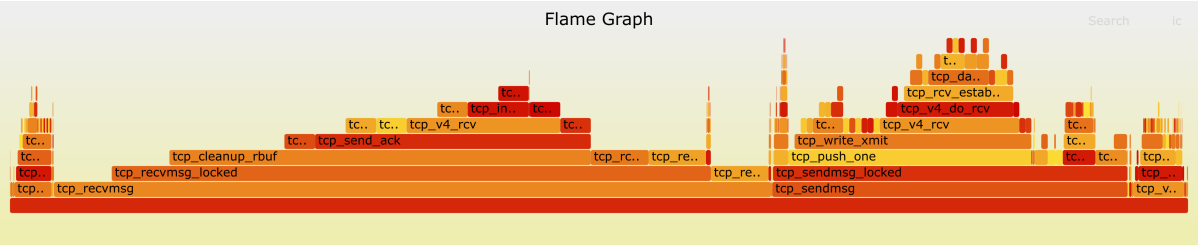

Figure 6.8: Flame graph of global tracing results for TCP-related kernel functions. Since logs from different CPUs are interleaved in the Ftrace tracing logs, some function call relationships may contain minor errors after being rendered as a flame graph, but it is still helpful for analyzing issues.

to perform a global tracing of the system and set up tracing of all TCP-related kernel functions. After obtaining the tracing logs, the engineers rendered them into a flame graph [37] to analyze this performance issue, as shown in Figure 6.8.

However, since many programs from other tenants on the cloud servers are using the TCP protocol to communicate, the engineers did not find any noteworthy anomalies in the system-wide flame graph. The main TCP behaviors in the flame graph were the common `tcp_recvmsg()` and `tcp_sendmsg()` kernel functions.

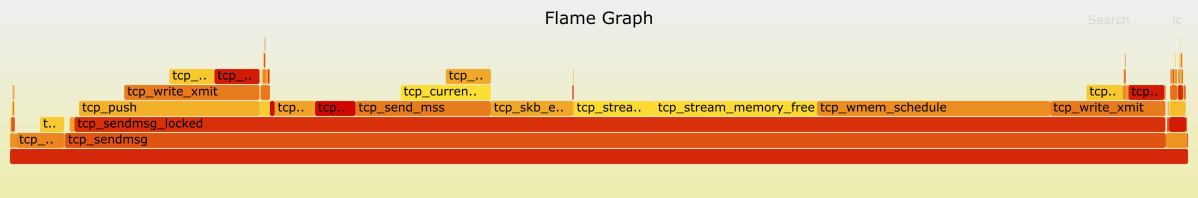

Figure 6.9: Flame graph of tracing results after event filtering based on the PIDs of processes within the container.

Engineers decided to perform a more granular analysis, filtering the PID of processes to trace only the behavior of processes within the container, and then re-rendered the flame graph, as shown in Figure 6.9. However, the engineers still did not find the problem. The main behavior of the container was `tcp_sendmsg()`, which was sending data normally.

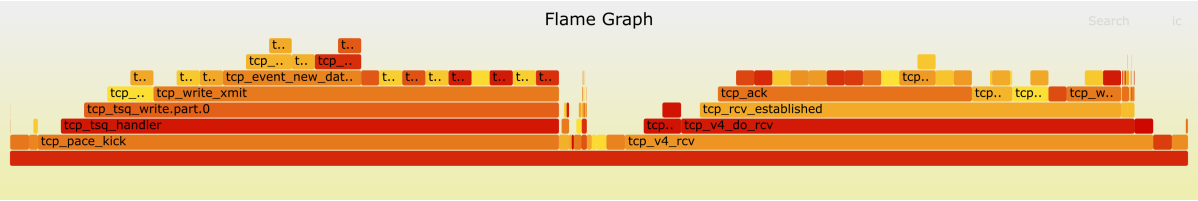

Figure 6.10: Flame graph of the tracing results obtained from the Ftrace instance within the container.

Engineers realized that PID filtering has limitations, resulting in it only being able to trace events in the process context, but not in the interrupt context or kernel thread context. Engineer decided to enter the container and use the Ftrace instance inside the container to perform system tracing, and obtained a new flame graph, as shown in Figure 6.10.

Engineers noticed an anomaly in the flame graph: nearly half of the container's TCP behavior was in `tcp_pace_kick()`. `tcp_pace_kick()` is called by timers, which is in soft interrupt contexts, so it could not be traced by the PID filtering method previously. This

is a kernel function in the TCP pacing feature [101] of the Linux kernel, used to control the rate limit and prevent data packets from being sent too quickly.

Engineers suspected that the cloud storage service program had incorrectly set the socket option `SO_MAX_PACING_RATE` [75]. After communicating with the media company's employees, the cause was finally discovered. The software development company forgot to remove some code that was added during software testing. After removing the relevant code, TCP performance returned to normal.

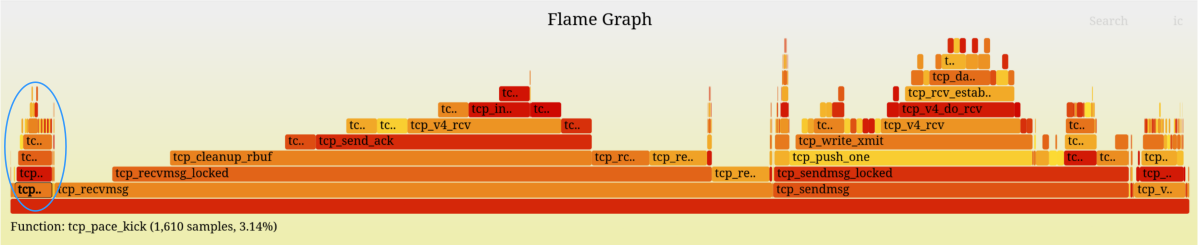

Figure 6.11: The flame graph of the global tracing results actually contains the part that can reveal the cause of the issue (circled by a blue circle on the left), but because it is only a small part of the flame graph, it is ignored by the engineers.

After understanding the cause, the engineers reviewed the initial global flame graph again and found that the `tcp_pace_kick()` part was in the flame graph (circled by a blue circle on the left), as shown in Figure 6.11. However, because this part only existed in a small corner of the flame graph, it was inadvertently overlooked by the engineers.

6.5 Summary

In real-world production scenarios, long-tail latency, system failures, and performance degradation are common and challenging issues [88, 33, 9, 32, 52]. In the case studies of this chapter, engineers resolved container issues through system tracing within container environments, including long-tail latency caused by dirty page accumulation (§6.1), TCP connection timeouts due to a full conntrack table (§6.2), long wait times for the first request caused by copy-on-write in overlayfs (§6.3), and poor TCP performance due to incorrect socket option settings (§6.4).

In the first three case studies (§6.1, §6.2, §6.3), engineers faced a challenging situation where the application behaves normally in the container, but the real issue is hidden in the system and cannot be effectively diagnosed in user space. To uncover system-level issues in production environments, engineers obtained critical container system-level runtime information by performing system tracing within the container. This critical information helped the engineers pinpoint the cause of the issues and ultimately resolve the issues they encountered.

If container tenants are unable to use system tracing within the container, the tenant needs to request assistance from the cloud provider to obtain system-level runtime information about the container. Tenants need to engage in extensive communication with cloud providers to describe the issues they encounter, and require cloud provider system administrators to conduct system tracing when anomalies may occur to obtain the necessary information. This is inefficient and inconvenient for tenants, and requires cloud providers to bear high support service costs.

The fourth case study (§6.4) illustrates another type of challenging situation where engineers can perform global system tracing and trace all system events on the system, but numerous irrelevant events obscure the actual cause. Engineers can gain different insights from a system-wide perspective and a more focused perspective within the container. This case study also demonstrates the limitations of currently widely used PID-based event filtering methods, which cannot effectively trace container-related events in interrupt contexts and kernel thread contexts.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

The shift of the cloud computing industry to containerization has exposed the inadequacies of current system tracing infrastructure in modern operating systems. Although the system tracing infrastructure in operating systems is already powerful and flexible, the support it can provide to tenants in multi-tenant container environments is limited, which causes inconvenience to tenants. The TraceNS proposed in this study aims to bridge this gap and is the first approach to provide secure and efficient system tracing in containers. The novel system operation partition context approach in TraceNS provides operation-global partition contexts for system operations, enabling system events triggered during system operations to be partitioned and isolated in the first place and delivered only to the corresponding tracing instances, rather than being filtered when the tenant processes the system events. As shown in the evaluation in this study, TraceNS can provide sufficient container-related system event coverage and security (system event isolation and sensitive event information hiding) while maintaining low container runtime overhead and low container creation overhead.

This study uses case studies to demonstrate how TraceNS enhances the understanding of container behavior in multi-tenant container environments and helps developers to analyze, debug, and optimize containers. Previously, these tasks required tenants to complete with the assistance of cloud providers, but with TraceNS, tenants can complete them themselves in their own containers. This is more convenient and efficient for tenants, and it also enhances their understanding of their containers and the transparency of their container behavior. For cloud providers, tenants can analyze common performance issues and anomalies on their own without the need for cloud providers to provide support, which reduces the cost of support services.

Previously, system observability typically meant observing the behavior of the system from a global (overall) perspective. This study demonstrates that the behavior of a system can be observed in a more granular way from the perspective of a container. Breaking down complex systems into smaller, more easily analyzed units, thus avoiding being overwhelmed by a large number of irrelevant events from the perspective of the overall system. This study uses a case study to demonstrate that a large number of irrelevant events from the global system perspective can distract engineers and cause them to overlook the actual problems. Observing the interactions and interferences between small units (containers) from a fine-grained perspective is an interesting new research direction worth exploring that this study opens up. New, fine-grained observation paradigms and new perspec-

tives on system observation can provide scientists and engineers with new insights and understandings of system behavior in high-density and complex system environments in academic research and industry.

7.2 Future Work

This study introduces TraceNS, which bridges the gap in system tracing that is not available within containers, and demonstrates the possibility of partitioning system events (system operations). However, TraceNS still has some limitations. This section discusses future work of this study, including the current limitations of TraceNS and the potential future research directions that this study opens up.

7.2.1 Multi-user Environment Tracing (Non-container)

In modern operating systems, container environments are not the only scenario where multiple tenants (users) share the same operating system kernel; this is also the case in ordinary multi-user environments. Multi-user environments are commonly found on internal servers in companies or universities. Employees/students can log in to a remote Linux host via SSH and share the system environment. Resource isolation in multi-user environments is typically achieved through user permissions. Ordinary users cannot access other users' directories, nor can they modify system directories or install new software.

In such multi-user environments, system tracing is typically not available to ordinary users. Because system tracing is global, it can trace the behavior of other users' processes and leak their private data. However, the unavailability of system tracing limits users' ability to analyze their applications through system tracing. For example, employees need to write programs on a multi-user shared Linux server to complete development tasks, but they cannot analyze system-level performance issues and anomalies in their programs through system tracing.

The approach proposed in this study for establishing partition contexts for system operations (system tracing) is a universal approach that can be applied to multi-user environments. Exploring approaches for providing system tracing in multi-user (non-container) scenarios, and providing an isolated view of system tracing for each user, is a future research direction of this study.

7.2.2 Integration with Novel Tracing Technologies

In the TraceNS prototype implementation of this study, integration with Ftrace has been achieved, providing isolated multi-view system tracing through pseudo file system interfaces. However, TraceNS currently provides limited support for eBPF-based system tracing [28]. This is not because the system event isolation provided by TraceNS is not universal, but because eBPF programs are too flexible. eBPF programs are turing complete. After JIT, they are equivalent to native binary code, which can dereference pointers in the kernel, access data structures in the kernel, and call eBPF helpers and kfuncs [29].

This means that although TraceNS can restrict only system events belonging to the container to reach the eBPF program, the eBPF program can jump to other objects related to the system event by dereferencing the kernel object pointer in the system event parameter. A well-constructed eBPF program can access kernel objects belonging to other containers through a series of pointer dereferences. Currently, eBPF programs can only be

used with administrator privileges or with `CAP_BPF`, which is secure. However, eBPF programs are clearly insecure for system tracing within containers in multi-tenant container environments.

This study will attempt to prevent eBPF programs from calling sensitive eBPF helpers or eBPF kfuncs, limit the scope of pointer dereferencing, and improve the eBPF verifier so that it only allows more restricted eBPF programs to be loaded in scenarios where the system tracing within containers. Exploring how to further restrict eBPF programs to adapt them to scenarios involving system tracing within containers is a future research direction of this study.

7.2.3 Predefined Events and Kernel Function Coverage

In the evaluation of TraceNS in this study, the coverage of predefined system events in the kernel subsystems and the security of the covered events were evaluated. This study will evaluate predefined system events in container-related kernel subsystems that have not yet been evaluated due to time constraints in the future.

The current Linux kernel supports tracing almost every kernel function in the kernel, not just predefined system events. Evaluating kernel function coverage is a difficult challenge because there are currently more than 70,000 traceable kernel functions in the Linux kernel. Triggering such a large number of kernel functions for coverage evaluation is no longer feasible through manually written workloads to trigger events.

Exploring automated methods such as fuzzing [103] to automatically execute system calls and try various possible parameter combinations to trigger as many kernel functions as possible for kernel function coverage evaluation is a future research direction for this study.

7.2.4 Noisy Neighbors in Cloud Environments

In shared cloud environments, a common and classic issue is "noisy neighbors". Since hardware resources are shared in cloud environments, a tenant's use of resources may potentially affect other tenants on the same machine. Cgroups [55] has already solved this issue to some extent, enabling container creators to limit the maximum CPU share, memory space, disk I/O, network bandwidth, etc. that a container can use. However, cgroups still cannot completely solve problems such as CPU cache contention, kernel lock contention, disk I/O tail latency, etc.

As shown in the evaluation of TraceNS in this study, the "noisy neighbor" issue can also arise when multiple tenants are tracing simultaneously. For example, when 100 containers simultaneously trace the system events on the same network packet processing code path, it can lead to a decrease in network performance, which is more pronounced in high-bandwidth, low-latency scenarios such as data centers. In high-density container deployment scenarios (such as 1000 containers on the same machine, potentially leading to numerous contentions), or in latency-sensitive high-performance scenarios, the impact of a tenant on other tenants (even a small impact) becomes a concern and needs to be mitigated as much as possible.

Exploring how to minimize the impact of "noisy neighbors" caused by system tracing in high-density container deployment scenarios and latency-sensitive scenarios, so that in-container system tracing can be better applied in these scenarios, is a future research direction of this study.

7.2.5 Distributed, Large-scale, High-density Environments

Due to limitations in the experimental environment and time constraints, this study has not yet been conducted experiments in distributed, large-scale, high-density system environments. In real-world container deployments, a tenant's containers may be deployed across multiple machines, multiple data centers, or even multiple data centers spanning multiple geographic locations (serving customers in different geographic locations). A tenant may deploy a large number of containers, forming a robust microservice architecture that is large-scale, distributed, and multi-replicated. A tenant may deploy containers on multiple different cloud providers to avoid single-provider anomalies. A cloud provider may deploy thousands of different tenant containers at high density on a single machine.

All of these will bring new challenges and opportunities to the container system tracing. In large-scale distributed environments, container system tracing faces the challenge of aggregating, synchronizing, and normalizing a large number of system tracing logs from different container nodes. In high-density environments, container system tracing faces the challenge of dealing with noisy neighbors, as mentioned in §7.2.4. On the other hand, the new system observation paradigm by treating containers as small units will bring new opportunities to study the interactions and interferences between containers in high-density system environments and large-scale distributed system environments.

Exploring the new challenges and opportunities presented by container system tracing in distributed, large-scale, and high-density system environments, and aiming to enhance the understanding of container behavior in these environments, is the future research direction of this study.

Bibliography

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[2] Amazon. What is AWS Lambda? `https://docs.aws.amazon.com/lambda/latest/dg/welcome.html`, 2026. Accessed on June 8, 2026.

[3] Akshat Aranya. Tracefs: A File System to Trace Them All. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[4] Jens Axboe. Flexible I/O Tester. `https://github.com/axboe/fio`, 2026. Accessed on June 8, 2026.

[5] Ivan Babrou. Tracing System CPU on Debian Stretch. `https://blog.cloudflare.com/tracing-system-cpu-on-debian-stretch/`, 2018. Accessed on June 8, 2026.

[6] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 198–212, New York, NY, USA, 1991. Association for Computing Machinery.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[8] Rabindra K. Barik, Rakesh K. Lenka, K. Rahul Rao, and Devam Ghose. Performance analysis of virtual machines and containers in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1204–1210, 2016.

[9] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[10] S.P. Bhattacharya and V. Apte. A Measurement Study of the Linux TCP/IP Stack Performance and Scalability on SMP systems. In *2006 1st International Conference on Communication Systems Software & Middleware*, pages 1–10, 2006.

[11] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace driven analysis of write caching policies for disks. *SIGMETRICS Perform. Eval. Rev.*, 21(1):13–23, June 1993.

[12] Shawn Bohrer and Bastien Dhiver. A debugging story: corrupt packets in AF_XDP; a kernel bug or user error? https://blog.cloudflare.com/a-debugging-story-corrupt-packets-in-af_xdp-kernel-bug-or-user-error/, 2023. Accessed on June 8, 2026.

[13] bpftrace. bpftrace. https://github.com/bpftrace/bpftrace, 2025. Accessed on June 8, 2026.

[14] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *2004 USENIX Annual Technical Conference (USENIX ATC 04)*, Boston, MA, June 2004. USENIX Association.

[16] cilium. cilium. https://cilium.io/, 2026. Accessed on June 8, 2026.

[17] Google Cloud. Observability in Google Cloud. https://docs.cloud.google.com/stackdriver/docs, 2025. Accessed on June 8, 2026.

[18] Henrique Zanela Cochak, Guilherme Piêgas Koslovski, Maurício Aronne Pillon, and Charles Christian Miers. RunC and Kata runtime using Docker: a network perspective comparison. In *2021 IEEE Latin-American Conference on Communications (LATINCOM)*, pages 1–6, 2021.

[19] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 16–22, New York, NY, USA, 2024. Association for Computing Machinery.

[20] crun. crun. https://github.com/containers/crun, 2026. Accessed on June 8, 2026.

[21] Mathieu Desnoyers and Michael Jeanson. Tracer Namespaces. https://lpc.events/event/16/contributions/1237/, 2022. Accessed on June 8, 2026.

[22] Docker. What is Docker? https://docs.docker.com/get-started/docker-overview/, 2026. Accessed on June 8, 2026.

[23] The Linux Kernel Documentation. ftrace - Function Tracer. https://docs.kernel.org/trace/ftrace.html, 2025. Accessed on June 8, 2026.

[24] The Linux Kernel Documentation. Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html, 2025. Accessed on June 8, 2026.

[25] The Linux Kernel Documentation. Linux Tracing Technologies Guide. https://docs.kernel.org/trace/index.html, 2025. Accessed on June 8, 2026.

[26] The Linux Kernel Documentation. Overlay Filesystem. https://docs.kernel.org/filesystems/overlayfs.html, 2025. Accessed on June 8, 2026.

[27] The Linux Kernel Documentation. Uprobe-tracer: Uprobe-based Event Tracing. https://docs.kernel.org/trace/uprobetracer.html, 2025. Accessed on June 8, 2026.

[28] eBPF Docs. Program type BPF_PROG_TYPE_TRACING. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_TRACING/, 2025. Accessed on June 8, 2026.

[29] eBPF Documentation. What is eBPF? https://ebpf.io/what-is-ebpf/, 2025. Accessed on June 8, 2026.

[30] Nicolas Ehrman. What is a container runtime? https://www.wiz.io/academy/container-security/container-runtimes, 2026. Accessed on June 8, 2026.

[31] Falco. The Falco Project. https://falco.org/docs/, 2026. Accessed on June 8, 2026.

[32] Zihao Fan, Enge Song, Bo Jiang, Yang Song, Yuke Hong, Bowen Yang, Yilong Lv, Yinian Zhou, Junnan Cai, Chao Wang, Yi Wang, Yehao Feng, Dian Fan, Ye Yang, Shize Zhang, Xiaoqing Sun, Jianyuan Lu, Xing Li, Jun Liang, Biao Lyu, Zhigang Zong, and Shunmin Zhu. Understanding the Long Tail Latency of TCP in Large-Scale Cloud Networks. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*, APNET '25, page 106–113, New York, NY, USA, 2025. Association for Computing Machinery.

[33] Peter Garraghan, Paul Townend, and Jie Xu. An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120, 2014.

[34] Mohamad Gebai and Michel R. Dagenais. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Comput. Surv.*, 51(2), March 2018.

[35] Grafana. About Grafana. https://grafana.com/docs/grafana/latest/introduction/, 2025. Accessed on June 8, 2026.

[36] Brendan Gregg. Viewing the Invisible. https://www.brendangregg.com/blog/2011-06-27/viewing-the-invisible.html, 2011. Accessed on June 8, 2026.

[37] Brendan Gregg. Blazing performance with flame graphs. In *27th Large Installation System Administration Conference*. USENIX Association, 2013.

[38] Brendan Gregg. Slack's Secret STDERR Messages. https://www.brendangregg.com/blog/2021-08-27/slack-crashes-secret-stderr.html, 2021. Accessed on June 8, 2026.

[39] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *ACM Trans. Comput. Syst.*, 30(3), August 2012.

[40] Open Container Initiative. Open Container Initiative Runtime Specification. https://github.com/opencontainers/runtime-spec, 2025. Accessed on June 8, 2026.

[41] Michael Jeanson. LTTng: The road to container awareness. https://archive.fosdem.org/2018/schedule/event/containers_lttng/, 2018. Accessed on June 8, 2026.

[42] Michael Jeanson. A follow-up on LTTng container awareness. https://archive.fosdem.org/2019/schedule/event/containers_lttng/, 2019. Accessed on June 8, 2026.

[43] Ann Mary Joy. Performance comparison between Linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, 2015.

[44] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[45] Yordan Karadzhov. Container tracing. `https://lpc.events/event/11/contributions/1104/`, 2021. Accessed on June 8, 2026.

[46] Kata. Kata Containers. `https://katacontainers.io/`, 2026. Accessed on June 8, 2026.

[47] KernelShark. KernelShark. `https://kernelshark.org/Documentation.html`, 2026. Accessed on June 8, 2026.

[48] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[49] Joanna Kosińska, Bartosz Baliś, Marek Konieczny, Maciej Malawski, and Sławomir Zieliński. Toward the Observability of Cloud-Native Applications: The Overview of the State-of-the-Art. *IEEE Access*, 11:73036–73052, 2023.

[50] Kubernetes. Overview. `https://kubernetes.io/docs/concepts/overview/`, 2026. Accessed on June 8, 2026.

[51] Rakesh Kumar and B Thangaraju. Performance Analysis Between RunC and Kata Container Runtime. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–4, 2020.

[52] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[53] Linux manual page. tc-netem(8). `https://man7.org/linux/man-pages/man8/tc-netem.8.html`, 2011. Accessed on June 8, 2026.

[54] Linux manual page. capabilities(7). `https://man7.org/linux/man-pages/man7/capabilities.7.html`, 2025. Accessed on June 8, 2026.

[55] Linux manual page. cgroups(7). `https://man7.org/linux/man-pages/man7/cgroups.7.html`, 2025. Accessed on June 8, 2026.

[56] Linux manual page. clone(2). `https://man7.org/linux/man-pages/man2/clone.2.html`, 2025. Accessed on June 8, 2026.

[57] Linux manual page. mkdir(2). `https://man7.org/linux/man-pages/man2/rmdir.2.html`, 2025. Accessed on June 8, 2026.

[58] Linux manual page. mkdir(2). `https://man7.org/linux/man-pages/man2/mkdir.2.html`, 2025. Accessed on June 8, 2026.

[59] Linux manual page. mount(2). `https://man7.org/linux/man-pages/man2/mount.2.html`, 2025. Accessed on June 8, 2026.

[60] Linux manual page. move_mount(2). https://man7.org/linux//man-pages/man2/move_mount.2.html, 2025. Accessed on June 8, 2026.

[61] Linux manual page. namespaces(7). https://man7.org/linux/man-pages/man7/namespaces.7.html, 2025. Accessed on June 8, 2026.

[62] Linux manual page. network_namespaces(7). https://man7.org/linux/man-pages/man7/network_namespaces.7.html, 2025. Accessed on June 8, 2026.

[63] Linux manual page. open_tree(2). https://man7.org/linux/man-pages/man2/open_tree.2.html, 2025. Accessed on June 8, 2026.

[64] Linux manual page. pid_namespaces(7). https://man7.org/linux/man-pages/man7/pid_namespaces.7.html, 2025. Accessed on June 8, 2026.

[65] Linux manual page. seccomp(2). https://man7.org/linux/man-pages/man2/seccomp.2.html, 2025. Accessed on June 8, 2026.

[66] Linux manual page. setns(2). https://man7.org/linux/man-pages/man2/setns.2.html, 2025. Accessed on June 8, 2026.

[67] Linux manual page. socket(2). https://man7.org/linux/man-pages/man2/socket.2.html, 2025. Accessed on June 8, 2026.

[68] Linux manual page. trace-cmd(1). https://man7.org/linux/man-pages/man1/trace-cmd.1.html, 2025. Accessed on June 8, 2026.

[69] Linux manual page. unshare(2). https://man7.org/linux/man-pages/man2/unshare.2.html, 2025. Accessed on June 8, 2026.

[70] Linux manual page. _exit(2). https://man7.org/linux/man-pages/man2/exit.2.html, 2026. Accessed on June 8, 2026.

[71] Linux manual page. exit_group(2). https://man7.org/linux/man-pages/man2/exit_group.2.html, 2026. Accessed on June 8, 2026.

[72] Linux manual page. open(2). https://man7.org/linux/man-pages/man2/open.2.html, 2026. Accessed on June 8, 2026.

[73] Linux manual page. posix_fadvise(2). https://man7.org/linux/man-pages/man2/posix_fadvise.2.html, 2026. Accessed on June 8, 2026.

[74] Linux manual page. rt_sigreturn(2). https://man7.org/linux/man-pages/man2/rt_sigreturn.2.html, 2026. Accessed on June 8, 2026.

[75] Linux manual page. tc-fq(8). https://man7.org/linux/man-pages/man8/tc-fq.8.html, 2026. Accessed on June 8, 2026.

[76] Linux manual page. veth(4). https://man7.org/linux/man-pages/man4/veth.4.html, 2026. Accessed on June 8, 2026.

[77] Linux manual page. vhangup(2). https://man7.org/linux/man-pages/man2/vhangup.2.html, 2026. Accessed on June 8, 2026.

[78] Linux manual page. write(2). https://man7.org/linux/man-pages/man2/write.2.html, 2026. Accessed on June 8, 2026.

[79] Sun Microsystems. Dynamic Tracing Guide - Introduction. https://illumos.org/books/dtrace/chp-intro.html, 2008. Accessed on June 8, 2026.

[80] Gabriele Mini. *OS-level virtualization with Linux containers: process isolation mechanisms and performance analysis of last generation container runtime*. PhD thesis, Politecnico di Torino, 2020.

[81] ESnet: Energy Sciences Network. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. `https://github.com/esnet/iperf`, 2026. Accessed on June 8, 2026.

[82] OpenTelemetry. What is OpenTelemetry? `https://opentelemetry.io/docs/what-is-opentelemetry/`, 2025. Accessed on June 8, 2026.

[83] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, page 15–24, New York, NY, USA, 1985. Association for Computing Machinery.

[84] Parca. Parca. `https://www.parca.dev/docs/overview/`, 2026. Accessed on June 8, 2026.

[85] perf. Linux kernel profiling with perf. `https://perfwiki.github.io/main/tutorial/`, 2026. Accessed on June 8, 2026.

[86] Pixie. Pixie. `https://github.com/pixie-io/pixie`, 2026. Accessed on June 8, 2026.

[87] Podman. What is Podman? `https://docs.podman.io/en/latest/`, 2026. Accessed on June 8, 2026.

[88] Rahul Potharaju and Navendu Jain. When the network crumbles: an empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[89] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *LISA*, volume 4, pages 241–254, 2004.

[90] Prometheus. What is Prometheus? `https://prometheus.io/docs/introduction/overview/`, 2025. Accessed on June 8, 2026.

[91] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, San Diego, CA, June 2000. USENIX Association.

[92] runc. runc. `https://github.com/opencontainers/runc`, 2026. Accessed on June 8, 2026.

[93] Sachchidanand Singh and Nirmala Singh. Containers & Docker: Emerging roles & future of Cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 804–807, 2016.

[94] Jakub Sitnicki. Conntrack turns a blind eye to dropped SYNs. `https://blog.cloudflare.com/conntrack-turns-a-blind-eye-to-dropped-syns/`, 2021. Accessed on June 8, 2026.

[95] Terin Stock. Lost in transit: debugging dropped packets from negative header lengths. `https://blog.cloudflare.com/`

lost-in-transit-debugging-dropped-packets-from-negative-header-lengths/, 2023. Accessed on June 8, 2026.

[96] Tetragon. Overview. https://tetragon.io/docs/overview/, 2026. Accessed on June 8, 2026.

[97] Tigera. Calico. https://github.com/projectcalico/calico, 2026. Accessed on June 8, 2026.

[98] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, page 133–142, New York, NY, USA, 2010. Association for Computing Machinery.

[99] Steven Vaughan-Nichols. What is Docker and why is it so darn popular? https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/, 2018. Accessed on June 8, 2026.

[100] Werner Vogels. File system usage in Windows NT 4.0. *SIGOPS Oper. Syst. Rev.*, 33(5):93–109, December 1999.

[101] Michael Welzl. TCP Pacing in the Linux Kernel. In *2025 International Conference on Computing, Networking and Communications (ICNC)*, pages 133–139, 2025.

[102] youki dev. youki. https://github.com/youki-dev/youki, 2026. Accessed on June 8, 2026.

[103] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.*, 54(11s), September 2022.