ISOLATION AS A QUANTIFIABLE RESOURCE IN MODERN LIGHTWEIGHT VIRTUALIZATION

by

Anjali

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: 08/03/2026

The dissertation is approved by the following members of the Final Oral Committee:

Michael M. Swift, Professor, Computer Sciences
Shivaram Venkataraman, Associate Professor, Computer Sciences
Rahul Chatterjee, Associate Professor, Computer Sciences
Joshua San Miguel, Associate Professor, Electrical & Computer Engineering
Ziqiao Zhou, Senior Researcher, Microsoft Research

तू धूप है, छम से बिखर
तू है नदी, ओ बेख़बर
बह चल कहीं, उड़ चल कहीं
दिल खुश जहाँ, तेरी तो मंज़िल है वहीं

You are the sunshine, go scatter yourself everywhere
You are a river, unaware of your own flow
Flow somewhere, fly somewhere
Where your heart is happy, there lies your destination.

— "Kholo Kholo", *Taare Zameen Par (2007)*; lyrics by Prasoon Joshi
English translation mine

To my parents, Seema and Vijay, for their tireless hard work and sacrifices so that my brother and I could have opportunities they never had.

Acknowledgments

I came to grad school without any plan, and I'm glad it turned out in a way I never expected. Many people guided me during this journey, in moments that, when I reflect on them, feel like important events in my life.

During my master's, I knew I had to give research a try. When I was looking for an advisor, I assumed Mike worked on architecture, and I hadn't even thought of talking to him – until a senior student advised me to, saying, "He does everything systems." I am very glad that I took that advice and emailed Mike about an independent study; after that, everything fell into place.

So first and foremost, I want to thank Mike Swift for giving me the chance to do research – something I never thought I would do. Among the many things I have learned from him over the years, the most important is the art of telling a story through research. I never thought research could tell a story, but like everything else in life, every research project is a story of new ideas, insights, and discoveries. I am also grateful to Mike for his patience when things took longer than expected, and for his support in the choices I made throughout grad school.

I want to thank all my committee members: Shivaram Venkataraman, Rahul Chatterjee, Joshua San Miguel, and Ziqiao Zhou. Shivaram has always been supportive, and his advice and feedback have greatly improved this work. I also had the pleasure of working with him as a teaching assistant for CS 537, where I learned a great deal about teaching and managing

a class. Rahul was always available for feedback and discussion, and I am grateful for his time and advice. Ziqiao was my mentor during one of my internships at Microsoft Research, and I am very happy she agreed to be on my committee. She inspired me throughout my internship, and I still aspire to her depth of low-level systems knowledge. Finally, thanks to Joshua for agreeing to serve on my committee and for his time and consideration of this work.

I also want to thank Tyler Caraza-Harter for being a coauthor on my first paper. His insights and help were invaluable, especially as I was finding my footing in research, and I am grateful to have worked with him.

I am grateful to have shared my PhD journey with members of the SCAIL research group, past and present. Nikhita Kunati was a great help during my PhD application process, and I thank her for her encouragement during that time. Mark Mansi and Yanfang Le were great first officemates; watching them work hard gave me the confidence and excitement I needed as a junior grad student. I especially want to thank Mark for his technical help and guidance when I was starting. Bijan Tabatabai and Sujay Yadalam were always available for feedback on research, and I value their advice immensely. Many thanks also to Hayden Coffey, Ashwin Poduval, and Yusen Liu for the discussions, feedback, and conversations that helped me along the way. To my peers in the broader systems group: Anthony, Vinay, Jing, Sambhav, Kostas, Abby, and many others, thank you for the conversations and discussions during reading groups and random hallway chats.

I also want to thank Chloe Alverti and Aishwarya Ganesan for being available for advice as I navigated my PhD and my decision to pursue academia. Talking through these choices with them helped me think more clearly at moments when I was unsure, and I am grateful for their guidance.

One of the most fruitful things I did besides research was running WACM with Suchita Pati during the pandemic. When everyone was struggling in their own way, having a sense of community was very important, and I am grateful for the opportunity to help and

organize events in that time of uncertainty. Organizing things then had its own layer of difficulty, but it was a rewarding experience.

I have been lucky to be surrounded by good friends throughout this journey. Thanks to all my friends in Madison for the hangout sessions and trips around Madison: Ojaswita, Sushma, Jacey, Sonia, Anthony, Suchita, Varsha, Naman, Sambhav, Sujay, Vinay, Chris, Siddharth, Soumya, and Varun. I especially want to thank Ojaswita and Sushma for being there for me during a difficult period in my PhD, always willing to listen and offer their support.

I also want to thank my friends outside of Madison. Thank you to Mehul, Madhu, Poonam, and Akansha for all the trips we have taken over the past years. Thanks to Ranjini for always being a call away whenever I needed her.

Last but not least, I would like to thank my family. My parents, Seema and Vijay, have always believed in all my decisions and have always been there for me. I also want to thank my brother, Abhishek, for exposing me to things related to computers from a young age, and specifically for encouraging me to get into open source development in undergrad, which proved to be a crucial point for me. I also want to thank my sister-in-law, Chaitali, for her constant warmth and support, and my niece, Renessa, for bringing joy to our lives.

Finally, to the city of Madison, thank you for being my home for these many years.

Contents

Contents vi

List of Tables x

List of Figures xii

Abstract xvi

1 Introduction 1

1.1 The Evolution of Virtualization for Cloud Infrastructure 1

1.2 The Isolation Selection Problem 3

1.3 Why is Isolation Difficult to Quantify? 4

1.4 Thesis Statement and Approach 6

1.4.1 Measuring Host-Kernel Code Coverage Across Isolation Platforms . 7

1.4.2 Identifying Object-Level Sharing in the Host Kernel 8

1.4.3 Modeling Isolation with Expectation of Risk (EoR) 10

1.4.4 Applying EoR to Agentic Workloads 11

1.5 Contributions 12

1.6 Overview 13

2 Background and Motivation 14

2.1 Isolation Platforms . . . 14

2.1.1 Linux Containers (LXC) . . . 15

2.1.2 KVM/QEMU . . . 16

2.1.3 gVisor . . . 16

2.1.4 Firecracker . . . 17

2.1.5 Other Platforms . . . 19

2.2 Balancing Isolation, Performance, and Efficiency . . . 20

2.3 Summary . . . 21

3 Blending Containers and Virtual Machines:

A Study of Firecracker and gVisor 22

3.1 Isolation Platform Comparison . . . 23

3.1.1 Methodology . . . 24

3.1.2 Security Policies . . . 24

3.1.3 Total Code Footprint . . . 26

3.2 CPU Workload . . . 27

3.2.1 Coverage Analysis . . . 27

3.2.2 Performance . . . 29

3.2.3 Insights . . . 29

3.3 Networking . . . 30

3.3.1 Coverage Analysis . . . 30

3.3.2 Code Differences . . . 31

3.3.3 Performance . . . 32

3.3.4 Insights . . . 35

3.4 Memory management . . . 36

3.4.1 Coverage Analysis . . . 36

3.4.2 Code Differences . . . 37

3.4.3 Performance . . . 37
3.4.4 Insights . . . 39
3.5 File access . . . 40
3.5.1 Coverage Analysis . . . 40
3.5.2 Performance . . . 41
3.5.3 Insights . . . 42
3.6 Summary . . . 43
3.7 Conclusion . . . 44

4 The Kernel Always Knows:
Understanding Isolation via Kernel Objects 45
4.1 Background and Motivation . . . 46
4.1.1 Isolation and Interference . . . 46
4.1.2 Performance vs. Efficiency . . . 48
4.1.3 Synchronization as an Identifier of Sharing . . . 49
4.2 Case Study . . . 52
4.3 LIMA Design . . . 55
4.3.1 LIMA Dynamic Tracer . . . 57
4.3.2 LIMA Static Analyzer . . . 58
4.4 Object Analysis Methodology . . . 60
4.4.1 Experimental platform . . . 61
4.4.2 Isolation Platforms . . . 62
4.4.3 Workloads . . . 62
4.4.4 Analysis Outputs . . . 64
4.5 Object Level Interference Analysis . . . 64
4.5.1 Platform Impact on Object Sharing . . . 65
4.5.2 Workloads' Sensitivity to Object Sharing Across Platforms . . . 73
4.5.3 Dominant Kernel Subsystems . . . 78

4.5.4 Performance Isolation . . . 78
4.6 Summary and Discussion . . . 79
4.7 Conclusions . . . 81

5 Out of the Woods, Into an Isolation Metric 82
5.1 A System Model of Isolation . . . 83
5.2 Breaking down Isolation, Numerically . . . 89
5.3 Properties of Expectation of Risk . . . 92
5.4 Discussion . . . 94
5.5 EoR Empirical Evaluation . . . 96
5.5.1 Identifying Interference Magnitude and Risk . . . 96
5.6 Conclusion . . . 101

6 Isolation in the Age of Agents: An EoR Case Study 102
6.1 Motivation: Tool-Aware Isolation . . . 103
6.2 Instantiating EoR for Agent Tasks . . . 106
6.3 Empirical Evaluation Methodology . . . 107
6.3.1 Tool-Call Replay Across Platforms . . . 107
6.3.2 Syscall and Execution-Time Profiling . . . 109
6.3.3 Estimating Shared-State Exposure . . . 109
6.3.4 EoR Calculation . . . 109
6.4 Results . . . 111
6.4.1 Slowdown under co-location . . . 111
6.4.2 Tool-level Slowdown . . . 112
6.4.3 EoR . . . 114
6.5 Discussion and Future Work . . . 118

7 Related Work 120
7.1 Lightweight Virtualization and Isolation Platforms . . . 120

7.2 Measuring Kernel Footprint and Virtualization Performance 122
7.3 Kernel Locks and Shared Kernel State 123
7.4 Measuring and Predicting Interference 123
7.5 Isolation and Security Metrics 124
7.6 Isolation for Agentic Workloads 125

8 Conclusions and Future Work 126
8.1 Summary 126
8.2 Lessons Learned 128
8.3 Future Work 131
8.4 Closing Remarks 133

Bibliography 134

List of Tables

3.1 Total number of system calls allowed out of 350 25
3.2 Union of line coverage across all workloads out of 806,318 total lines in the Linux kernel. 26
3.3 Round-trip time 35

4.1 Sharing across different platforms. 47
4.2 Top-5 locks and other locks for *lxc* and *fc* for *listdirs*. 52
4.3 Lock primitives probed by *LIMA* dynamic tracer. 55
4.4 Core functionalities of *LIMA*'s static analysis. 58
4.5 Average traces collected across workload events, interrupts, and other events across all workloads and platform combinations. 60
4.6 Microbenchmarks with their descriptions. 61
4.7 Cloud, serverless workloads and stress-ng stressors analyzed by *LIMA*. 61
4.8 Top shared locks across our collected traces. 65

5.1 Symbols used in the isolation system model. 84
5.2 Examples for calculating EoR for different resources. 94
5.3 Units used in calculating EoR. 96

6.1 Spearman rank correlation between EoR and measured corun slowdown (N=10), pooled across platforms and within each platform. *n.s.*: not significant ($p > 0.05$). 114

6.2 Median task-level EoR and measured corun slowdown by platform (N=10). . . 116

6.3 Spearman rank correlation between per-tool EoR and measured per-tool corun slowdown (N=10), using execution-time (Figure 6.3a) and syscall-time (Figure 6.3b) slowdown. Each within-platform correlation uses 14 tool categories. . 117

List of Figures

1.1 Interplay between isolation, performance, and resource efficiency across isolation platforms. Measuring isolation will lead to precise estimations of the tension between the three components. 3
1.2 Spectrum of OS functionality location in isolation platforms. 8

2.1 gVisor architecture 18
2.2 Firecracker architecture 18

3.1 CPU workload coverage 27
3.2 CPU workload 28
3.3 Network Workload Coverage 30
3.4 Hits in net/core/dev.c 31
3.5 Aggregate Network Bandwidth 32
3.6 Host network profile 33
3.7 Firecracker network profile 33
3.8 LXC network profile 34
3.9 gVisor network profile 35
3.10 Memory workload coverage 36
3.11 Total allocation time (without munmap) for 1GB 38

3.12 Total touch time (without munmap) for 1GB 38
3.13 Total allocation+unmap time for 1GB 39
3.14 Total touch time (with munmap) for 1GB 39
3.15 File write workload coverage 40
3.16 Write Throughput 41
3.17 Read Throughput 41

4.1 Examples of lock usage patterns in the Linux kernel. 47
4.2 Impact of interfering workloads on *listdirs* performance over time. 52
4.3 Overview of *LIMA* workflow. In the dynamic tracing phase, *LIMA* collects lock accesses across workloads, and the static analysis phase completes the lock-to-object mapping. 55
4.4 Figures 4.4a–4.4c show the median aggregate lock counts and figures 4.4d-4.4f show average access rates across platforms for each workload category. 66
4.5 FunctionBench total lock access rate by subsystems. Most workloads show high usage in *mm* and *fs* across platforms. 67
4.6 Cloud workloads total lock access rate by subsystems. Most workloads show high usage in *mm* and *fs* across platforms. 68
4.7 Performance impact on model training when run with varying stress-ng *stressors*. 78

5.1 Components of an isolation platform. 84
5.2 Environment, E_s of an isolated workload, s in the isolation system model. . . . 84
5.3 EoR and performance change for symmetrical workloads. 98
5.4 Pearson correlation for each FunctionBench (`s`) and stress-ng (`stressor`) pair. . 100
5.5 EoR and performance change for video processing. 100

6.1 Resource utilization and execution-time distribution per tool category across isolation platforms. 104
6.2 Task-level slowdown under co-location with N=10 neighboring workloads. . . 112

6.3 Tool-level slowdown under co-location with N=10 neighboring workloads. . . 113

6.4 Task-level EoR versus measured corun slowdown. 115

6.5 Tool-level EoR. 117

Abstract

Virtualization is the backbone of cloud computing, where mutually distrusting tenants share physical hardware. Originally used to run monolithic applications, the cloud architecture has evolved to support microservices. In recent years, a new architecture known as Serverless Computing, or function-as-a-service (FaaS), has become popular. Serverless computing eliminates the need for developers to manage servers and hardware. In this architecture, different logical components of a microservice run as separately scheduled functions.

Several virtualized solutions have also been proposed to meet the changing needs of the architecture and design of the cloud computing model in the last decade. All these solutions are built to meet the twin goals of a multi-tenant environment, *i.e.,* performance and isolation. To ensure performance, a workload must perform with low overhead and without slowdowns induced by co-located workloads competing for shared resources. Isolation requires mutually distrusting workloads to remain confined, unable to access, corrupt, or infer one another's data or execution through the shared host. These goals are in tension: stronger isolation boundaries typically impose greater performance or resource-efficiency costs, whereas weaker isolation can improve efficiency through increased resource sharing, but at the risk of interference.

Most of these solutions, like Google's gVisor or AWS Firecracker, follow a *Lightweight Virtualization* design by either stripping down functionalities of full system virtualization,

such as in a microVM, or by adding more functionalities in userspace to limit interactions with the shared host platform, such as in a Secure Container. We call these lightweight virtualized solutions *Isolation Platforms*. When workloads run in a mutually distrusting multi-tenant environment, a central goal is to limit interaction with the outside world while maintaining performance guarantees. As these platforms solve the problem of tightening the isolation boundaries by taking different approaches in design, it can be difficult for a developer to choose a platform that provides *enough* isolation for their workloads.

Isolation in computer systems keeps workloads apart so that the behavior of one workload does not affect other workloads. Perfectly isolating workloads in a cloud environment is a long-standing problem in systems [85, 65, 138, 69]: complete isolation can be achieved by running workloads on separate physical machines, *i.e.*, no sharing at all [77], but this wastes resources and removes major benefits of cloud computing. All of these isolation mechanisms follow the principle of reducing the attack surface to the shared host kernel. This is mostly achieved by limiting the system calls interface to the host. Although all these solutions promise an increase in isolation, there is no way to quantitatively measure how much isolation they *actually* provide to a given workload. This problem becomes even more important with the growing use of agentic AI systems, where agents may execute code that is dynamically generated, externally supplied, or difficult to inspect before execution. In this context, the question is no longer whether workloads should be isolated, but how they should be isolated, and which platform provides sufficient isolation for a given agent task.

This dissertation argues that isolation is not a fixed property of a platform, but a workload-dependent and quantifiable one, determined by how a workload interacts with shared host-kernel state. Conventionally, isolation is assessed qualitatively – virtual machines are presumed to be more isolated than containers – leaving no way to measure how much isolation a platform provides for a particular workload, or whether that isolation is sufficient. We study isolation across different platforms: Linux containers, Google's gVisor and AWS Firecracker microVMs, through different lenses. We develop methods and tools

to quantify isolation as a system property. We analyze how workloads interact with the host kernel – via kernel code paths and object usage – and formulate a quantitative isolation metric that enables comparing platforms based on their isolation guarantees. Finally, we apply this metric to agentic workloads, where tool execution creates diverse, unpredictable contact with shared host resources, making the choice of the *right* isolation platform both difficult and consequential.

In the first part, we analyze how different design choices impact performance cost and the use of Linux kernel services across different platforms. We evaluate the design of these platforms using kernel code coverage and microbenchmarks, comparing them to Linux containers and native Linux. Despite their minimal designs, the results show that both AWS Firecracker and Google's gVisor execute significantly more kernel code than native Linux. These findings show that identical workloads interact with the host kernel differently across platforms, underscoring the need for quantitative methods to evaluate isolation-efficiency trade-offs.

In the second part, we move from kernel code footprint to object-level sharing. We introduce *LIMA*, a tool combining dynamic tracing and static analysis to detect fine-grained object sharing and quantify synchronization frequency as a signal of interference. Using *LIMA*, we show that platform design shapes both the breadth and intensity of shared kernel-state access. We find that memory allocation and file-system journaling dominate cross-application interference. Linux containers expose broad sharing, gVisor mediates many interactions through a userspace kernel while still relying on host memory and filesystem paths, and Firecracker concentrates access on critical kernel objects. These results show that isolation is not binary, but quantifiable – defined by how platforms manage shared kernel data.

While the first two parts identify where and how workloads interact through shared kernel state, they do not by themselves provide a metric for comparing isolation. To address this, in the third part, we define a formal model for a cloud environment and develop an

isolation metric, Expectation of Risk (EoR). The goal of the metric is to allow a comparison of the relative amount of isolation a platform offers to an executing workload. We identify interference points (points where parallel workloads can interfere with each other, such as shared files and devices, concurrent memory accesses *etc.*) to quantify the amount of interference, assuming each point carries a unit of risk (a value of 0 means the interference point has no detrimental impact on workloads). Using these risk values and the interference magnitude (a quantity that quantifies *how much* a workload is interfered with through a shared resource) at each point, we calculate the *Expectation of Risk* (EoR).

In the final part, we apply EoR to agentic workloads. AI agents translate model decisions into action through tools that read files, spawn processes, execute code, and issue network requests, making tool execution a key point of contact with shared host resources—yet an agent's exact tool usage is not known in advance. We characterize agent tasks under co-location and show that they experience substantial and heterogeneous slowdown across isolation platforms. We instantiate EoR for shared host-kernel exposure using host-level syscall profiles and shared kernel locks, and show that task-level EoR correlates with measured co-run slowdown across platforms, providing a first step toward tool-aware selection of isolation platforms.

Together, these studies establish isolation as a measurable, workload-dependent property of modern virtualized systems, and provide the tools, metric, and empirical foundation for choosing and eventually designing isolation mechanisms based on quantified need rather than conventional wisdom.

— 1 —

Introduction

> One accurate measurement is worth a thousand expert opinions.
>
> *Rear Admiral Grace M. Hopper*

1.1 The Evolution of Virtualization for Cloud Infrastructure

The shared infrastructure that forms the backbone of modern computing has continually evolved to meet the needs of a rapidly changing computing landscape. The foundation of the modern cloud is built on this evolution, balancing: (1) the efficiency of sharing hardware among many tenants, (2) performance of running multiple workloads, and (3) the security of keeping those tenants apart. While early mainframe computers served multiple customers on shared infrastructure [57, 122], the move to mini and then microcomputers enabled the use of dedicated physical machines for each service. However, this led to inefficiencies from unused resources stranded on machines and high management costs.

The rise of virtualization [116, 44] provided a mechanism to keep the benefits of separate machines – unique operating system application installations – but reap the efficiencies of shared hardware running at higher utilization. While used early for server consolidation within an enterprise, virtualization is now the backbone of cloud computing, where mutu-

ally untrusting tenants share physical hardware from a third party. Shared infrastructure as we know today, along with providing resource utilization efficiency, requires on-demand creation, server management, and containment of the negative impact of malicious or malfunctioning applications.

As the cloud has shifted from monolithic applications toward microservices and, more recently, serverless functions, the unit of deployment has become smaller and its lifetime has shortened [127, 87, 75]. Full virtual machines, each with its own operating system, have become too heavyweight to launch a single short-lived function. Linux containers solved this goal with OS-level virtualization: built on kernel namespaces and control groups, they give each tenant a private view of files and resources while sharing a single kernel installation. Containers are fast and flexible, but because they share the same host kernel, the isolation they provide is bounded by that kernel's correctness, and a single kernel bug can compromise inter-tenant security [141, 150].

This gap has motivated a new class of *lightweight virtualization platforms* in the last decade that sit between containers and full virtual machines. AWS Firecracker [16] runs each workload in a stripped-down microVM with a complete guest kernel but a minimal device model, narrowing the interface to the host. Google gVisor [18] takes the opposite approach, interposing a user-space kernel (the *Sentry*, written in Go) that services most system calls itself, widening the software layer between the application and the host kernel. These platforms can be organized on an *isolation spectrum* of how much of the system functionality resides in the shared host kernel: from native Linux and containers at one end, through gVisor and Firecracker in the middle, to full virtualization at the other. We describe these platforms and this spectrum in detail in Chapter 2.

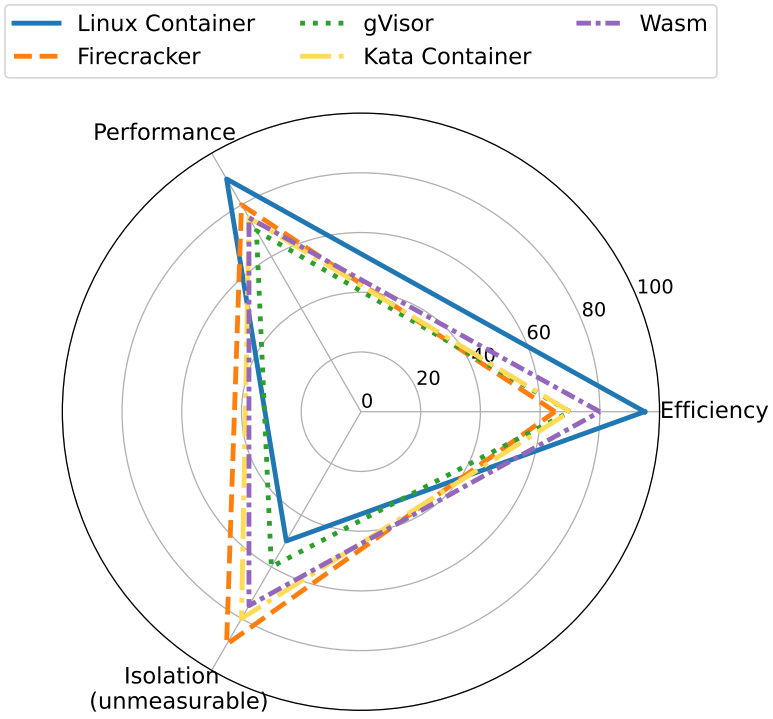


Figure 1.1: Interplay between isolation, performance, and resource efficiency across isolation platforms. Measuring isolation will lead to precise estimations of the tension between the three components.

1.2 The Isolation Selection Problem

These isolation platforms occupy different points on the isolation spectrum, and cloud providers expose them to developers as different service options [2, 4, 1, 6, 3, 7, 8, 9, 5] to provide different degrees of isolation. In AWS alone, a tenant can reserve an entire physical machine for complete isolation [4], run virtual machines under the Nitro hypervisor [9], deploy Firecracker microVMs [8, 6], launch Linux containers [5], or use confidential VMs for data isolation [7]. Each choice strikes a different balance between isolation, performance, and resource efficiency.

These three goals are in fundamental tension, as illustrated in Figure 1.1. More efficient platforms (*e.g.*, processes and containers) share more with the host, while stronger isolation platforms (*e.g.*, full VMs and microVMs) share less. Sharing improves utilization,

but it also creates opportunities for one workload to degrade another's performance (*e.g.*, by exhausting memory) or to compromise its security (*e.g.*, through unwanted access to shared state). Opting for more isolation typically costs efficiency or performance, because it demands more dedicated resources. Furthermore, platforms vary in the isolation they *provide*, but workloads also vary in the isolation they *need*: sensitive workloads (*e.g.*, payment processing) demand strong isolation, while latency-critical services [139] prioritize performance stability and batch jobs can tolerate variability.

The result is the *isolation selection problem*: given a workload and a set of co-located neighbors, *where* (which node) and *how* (which platform) to deploy it? While having many options is valuable, choosing the *right* one is difficult, because there is no standardized way to represent the trade-off in Figure 1.1, let alone optimize it. The selection problem is becoming more acute with evolving agentic AI systems, whose agents execute code that is dynamically generated, externally supplied, or difficult to inspect before execution–so the question is no longer whether to isolate a workload [47, 109, 144], but how, and which platform provides sufficient isolation for it.

1.3 Why is Isolation Difficult to Quantify?

The root of the selection problem is a measurement gap: although isolation platforms promise stronger boundaries, there is no standard way to quantify how much isolation a platform *actually* provides to a given workload. Without such a measure, deciding both *where* to place a workload and *how* to isolate it remains a complex task. A cloud scheduler may need to choose a node for a workload based on the behavior of already-running tenants, while a developer may need to choose between containers, secure containers, microVMs, full virtual machines, or stronger forms of isolation. In both cases, the decision depends on the amount of isolation the workload needs and the amount of isolation the platform can provide.

Several factors make quantifying isolation challenging. We identify three main dimensions of this measurement problem: isolation is *dynamic*, *system-specific*, and *context-dependent*. It is *dynamic* because the effective isolation of a workload changes with its environment. A workload may be well isolated when co-located with compute-heavy tasks, but may experience interference when placed next to workloads that stress the same filesystem, memory allocator, network stack, or device path. Isolation therefore cannot be evaluated only by studying a workload alone; it must be understood in the presence of the other workloads sharing the machine.

Isolation is also *system-specific*: the same high-level platform abstraction can hide very different system behavior. Scheduling decisions, memory allocation paths, filesystem metadata updates, I/O handling, and the way an isolation platform mediates access to shared kernel state all influence whether two workloads can affect each other. Two platforms may both appear to provide strong isolation at the deployment level, yet exercise different host-kernel paths and share different kernel objects underneath.

Finally, isolation is *context-dependent* because the meaning of *enough isolation* depends on the workload's goals. The same amount of interference may be acceptable for a batch job but unacceptable for a latency-critical service. Similarly, sharing that is harmless for an insensitive workload may be unacceptable for a security-sensitive workload, such as payment processing or workloads handling private data. The relevant question is therefore not simply whether a platform is isolated, but whether it provides sufficient isolation for a particular workload in a particular deployment context.

As a result, it remains difficult to accurately visualize the tradeoffs between isolation, performance, and efficiency shown in Figure 1.1. This gap emphasizes the need for a standard framework for defining and quantifying isolation that applies to diverse scenarios. Existing approaches to accurately measure interference tend to focus on specific use cases (*e.g.*, different approaches for each resource like storage, CPU, memory) [53, 65, 69, 151, 130, 121]. This limits their scope from being applied broadly, such as when cloud infrastructure

changes with dynamic scaling, new resource-sharing mechanisms, and new virtualization platforms, and evolving workloads with diverse characteristics. We bridge this gap by establishing a formal approach to isolation by defining and quantifying it through a metric.

A standard metric for isolation would enable several capabilities:

- *Choice of platform:* A metric allows comparison of expected isolation of a workload placement on different isolation platforms.
- *Choice of placement:* A metric allows automatic placement considering isolation by a scheduler that ensures maximum or a desired level of isolation.
- *Tradeoff with performance and efficiency:* A metric enables optimizing the tradeoffs between performance, efficiency, and isolation, and making better workload placement choices.
- *Choice of isolation degree:* A metric assists analysis and evaluation of the cost for selecting different isolation platforms [52, 44, 105, 36, 81, 25, 21, 120] for a given workload, *i.e.,* which platform provides *enough* isolation to the workload: *Do we need to use a microVM, or can the same isolation benefits be accomplished in a more efficient secure container?*
- *Design for isolation:* A metric allows platform designers to study the impact of different design choices (*e.g.,* coarse vs. fine-grained sharing) while accessing resources, enabling better design decisions.

1.4 Thesis Statement and Approach

This dissertation argues that *isolation is not a fixed property of a platform, but a workload-dependent, quantifiable property shaped by how workloads interact with shared host-kernel state.* A platform does not simply isolate or fail to isolate; it isolates a *particular workload* to

a *measurable degree* that depends on the kernel state that the workload touches and the neighbors it runs alongside.

We defend this claim through different complementary lenses, each building on the last and each contributing a method or tool for treating *isolation as a system property.*

1.4.1 Measuring Host-Kernel Code Coverage Across Isolation Platforms

Serverless computing is an execution model in which tenants provide lightweight *functions* to a platform provider, which is responsible for finding a host and launching them on that host. As public clouds are multi-tenant, serverless computing naturally seeks to run functions from multiple tenants on the same physical machine, which requires strong isolation between tenants. For example, AWS ran serverless functions in Linux containers inside virtual machines, with each virtual machine dedicated to functions from a single tenant [141]. In contrast, Azure multiplexed functions from multiple tenants on a single OS kernel in separate containers [141]. As a result, a kernel bug could compromise inter-tenant security on Azure but not AWS.

Recently, lightweight isolation platforms have been introduced as a bridge between containers and full system virtualization. Amazon Web Services (AWS) Firecracker [16] and Google's gVisor [18] both seek to provide additional defense and isolation in their respective serverless environments. Both are based on the principles of least privilege and privilege separation. Firecracker takes advantage of the security and workload isolation provided by traditional VMs but with much less overhead via lightweight I/O services and a stripped-down guest operating system. gVisor, in contrast, implements a new user-space kernel in Go to provide an additional layer of isolation between containerized applications and the host operating system.

In order to make sense of these new platforms and how they compare with containers and heavyweight system virtual machines, we can organize the platforms on a spectrum, illustrated in Figure 1.2. The position illustrates how much of their functionality resides in

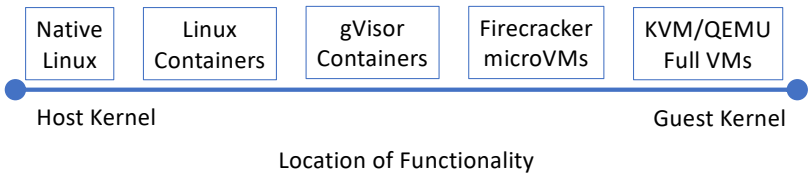


Figure 1.2: Spectrum of OS functionality location in isolation platforms.

the host operating system. At the left, the host operating system provides all functionality. With Linux containers (LXC), a user-mode daemon plays an important role in provisioning and configuring containers. gVisor sits in the middle, as it moves execution of most system calls out of the kernel into a user-mode *Sentry* process. Firecracker runs a lightweight virtual machine (*microVM*) with a complete guest kernel, but relies on the host operating system to securely isolate the Firecracker hosting process. Full virtualization, such as QEMU/KVM, moves most functionality out of the host kernel into either the guest operating system or the host QEMU process.

We perform the first fine-grained kernel code-coverage analysis of Linux containers, gVisor, and Firecracker, measuring *which* host-kernel code each platform executes and *how frequently*. Despite their minimal designs, both gVisor and Firecracker execute substantially *more* host-kernel code than native Linux for some workloads; for example, in the networking workload, gVisor executes up to 80K lines of host-kernel code and Firecracker executes around 56K. This shows that identical workloads exercise the host kernel very differently across platforms (Chapter 3).

1.4.2 Identifying Object-Level Sharing in the Host Kernel

Code coverage reveals how broadly each platform enters the host kernel, but not what workloads touch once inside it. Even highly isolated platforms ultimately depend on shared kernel resources, such as memory management, file systems, and schedulers, that are managed through shared kernel objects: kernel-resident data structures that hold system state and coordinate access to shared resources (*e.g.*, inodes, page allocators,

scheduling queues). While platforms employ OS-level (*e.g.*, cgroups, qdisc) and hardware-level (*e.g.*, cache partitioning) mechanisms to partition CPU, memory, and network bandwidth [53, 68, 69, 152, 88, 123], resource partitioning does not eliminate interference caused by shared access to kernel objects. This shared kernel state introduces invisible structural dependencies between workloads that can undermine isolation guarantees, manifesting as unpredictable performance behavior (*e.g.*, noisy neighbors) [50], or security vulnerabilities (*e.g.*, framing attacks [114] or denial of service [84]) that are difficult to reason about without visibility into how kernel-managed data structures are shared across platforms.

The key challenge in understanding object-level sharing lies in efficiently identifying shared structures within a large and complex kernel codebase. We leverage synchronization of shared state: by design, the operating system synchronizes access to shared kernel objects, so we use synchronization events as a proxy for potential object interference between workloads. A key insight of this approach is that object-level isolation, like the kernel coverage of the previous study, is not a static property of the platform but a dynamic function of the workload: workloads that rarely synchronize (e.g., CPU-bound tasks in user-space) interact with little shared kernel state and effectively operate on disjoint objects, achieving high isolation regardless of the underlying platform.

We develop a tool, LIMA (Lock Interference Mapping Analyzer), which combines dynamic tracing of kernel locking events with static analysis to map kernel locks to the underlying data structures they protect. We use LIMA to conduct a comparative study of the sharing and interference observed at the system software level when concurrent workloads execute on different isolation platforms. We show that isolation is shaped by platform design: Linux containers share a wide variety of objects with high intensity; Firecracker narrows this variety but accesses specific hot objects intensely; and gVisor shares across a moderate number of objects while maintaining lower acquisition rates. We identify memory allocation and file system journaling as the dominant sources of cross-application interference (Chapter 4).

1.4.3 Modeling Isolation with Expectation of Risk (EoR)

The two preceding studies establish that isolation depends on which kernel code and objects a workload exercises, and alongside whom – but they measure this exposure rather than express it as a single quantity that deployment decisions can act on. Determining a workload's isolation requires exactly this context: *What other workloads are co-located? What isolation platform is used for a workload? Does the workload access sensitive information, prioritizing isolation over performance?* A workload may be well isolated if it uses resources that the platform isolates well, but suffers interference when accessing poorly isolated ones.

This context-dependence makes the decision of where (node selection) and how (platform selection) to deploy a workload a rather complex task. Current approaches to balancing isolation with performance and efficiency include monitoring resource metrics (e.g., CPU, memory), measuring performance (e.g., latency, bandwidth), running stress tests [69, 101, 68, 53], and applying ML to historical workload data [61]. While informative, these methods only indirectly capture isolation, which is often treated as unmeasurable. Isolation is hard to quantify because it depends on dynamic factors (*e.g.*, co-running workloads), system-specific behavior (*e.g.*, scheduling), and context (*e.g.*, security vs. performance), and most existing techniques measure interference using a specific mechanism [53, 65, 69, 151, 130, 121], limiting their generality. To bridge this gap, we need a formal, universal metric for isolation that applies across platforms, workloads, and resource types.

This leads to the fundamental question: *Can we quantify isolation using a metric applicable to evolving cloud infrastructures?* Our core intuition is that if we can identify interference through shared resources in the system, we can formulate a way to measure it. Based on this insight, we propose a new isolation metric, *Expectation of Risk* (*EoR*), to quantify the isolation a workload gets in an environment. We define the key entities in a workload's environment, including co-running workloads and shared resources, and develop a framework that identifies where interference can occur, quantifies its magnitude at each interference point, and assigns an associated risk reflecting the severity of its potential impact. By combining

interference magnitude with associated risk at each interference point, we calculate EoR, allowing comparison of the relative isolation a workload gets under different environments and determination of which environment offers enough isolation for a given workload (Chapter 5).

1.4.4 Applying EoR to Agentic Workloads

Finally, we apply EoR to an emerging class of workloads for which the isolation selection problem is especially acute: AI agents. Agents combine models for reasoning with tools for action, enabling them to translate human intent into autonomous execution [74], and are increasingly deployed in multi-tenant environments [153, 48]. From a systems perspective, the tool layer [153, 92, 149, 125, 115] is where an agent's decisions become resource-consuming execution: tools read and write files, spawn processes, execute code, issue network requests, and interact with external state [153, 92, 149, 125, 115], making tool execution a major point of contact with shared OS resources whether tools run with the agent orchestrator [94] or through a remote multi-tenant tool server such as MCP [40, 56].

Agentic workloads sharpen the isolation dilemma of Section 1.2. They are interactive but not necessarily latency-sensitive, can have high utilization because agent think time is low and parallelism can be high, and heterogeneous, since an agent may invoke a wide range of tools, including tools not known when the environment was provisioned. Isolating each tool in a separate environment would provide fine-grained boundaries but adds startup latency and duplicated runtime state; in practice, an agent's tools are often executed within a common isolation environment, forcing systems to choose one boundary for a diverse and changing toolset. Moreover, an agent's exact sequence of tool invocations is often not known in advance — yet agent behavior is typically guided by a bounded set of accessible tools and task-level intent [115, 149], providing a basis for approximating how likely an agent is to interact with shared resources.

Building on this observation, we instantiate EoR to estimate the interference risk in-

duced by an agent's tool-mediated interactions with shared infrastructure, using host-level syscall profiles and the shared kernel locks identified by LIMA to model shared host-kernel exposure. Task-level EoR correlates with the co-location slowdown agents actually experience across platforms, providing a first step toward tool-aware runtime selection, where isolation risk can be combined with performance and efficiency goals (Chapter 6).

1.5 Contributions

This dissertation makes the following key contributions:

- **A fine-grained characterization of host-kernel use across isolation platforms.** We develop a code-coverage and microbenchmarking methodology and apply it to Linux containers, gVisor, and Firecracker, showing which kernel code each platform executes and at what frequency, and quantifying the resulting performance trade-offs across CPU, networking, memory, and file-system workloads. We find that lightweight platforms execute more kernel code than native Linux and that no single platform is best for all workloads (Chapter 3).

- **LIMA, a tool for measuring object-level isolation.** We build LIMA, which detects fine-grained kernel object sharing by combining dynamic tracing of kernel locking events with static analysis that maps locks to the data structures they protect. LIMA grounds isolation behavior in measurable kernel-object sharing rather than platform abstractions, enabling direct comparison of platforms (Chapter 4).

- **An object-level study of isolation across platforms.** Using LIMA, we collect and analyze kernel lock traces across a diverse set of workloads and isolation platforms. We show how Linux containers, Firecracker, and gVisor differ in the breadth and intensity of the kernel state they share, and identify memory allocation and file-system journaling as the dominant sources of cross-application interference (Chapter 4).

- **The Expectation of Risk (EoR) metric.** We formalize the entities of a workload's environment and define EoR, a metric that quantifies the isolation a workload receives by combining interference magnitude with risk at each shared interference point. EoR enables a direct quantitative comparison of the isolation that different environments and platforms provide to a given workload (Chapter 5).

- **A tool-aware case study of isolation for agentic workloads.** We characterize the resource usage and interference behavior of AI-agent tasks, whose tool-mediated execution makes them a demanding new class of multi-tenant workload, and instantiate EoR for shared host-kernel exposure using host-level syscall profiles and shared kernel locks. We show that agent tasks experience substantial and heterogeneous slowdown under co-location across isolation platforms, and that task-level EoR correlates with measured co-run slowdown, providing a first step toward tool-aware isolation platform selection (Chapter 6).

1.6 Overview

The remainder of this dissertation is organized as follows. Chapter 2 provides background on virtualization and the four isolation platforms we study, and develops the tension among isolation, performance, and efficiency that motivates this work. Chapter 3 analyzes how Firecracker, gVisor, and containers use host-kernel functionality through code coverage and microbenchmarks. Chapter 4 presents LIMA and our study of object-level kernel-state sharing across platforms. Chapter 5 defines the formal isolation model and the Expectation of Risk (EoR) metric, and evaluates it empirically. Chapter 6 applies EoR to agentic workloads, characterizing their interference behavior under co-location and evaluating EoR as a tool-aware isolation signal. Chapter 7 surveys related work across the three parts of the dissertation, and Chapter 8 concludes and discusses directions for future work. Each chapter includes its own relevant background.

— 2 —

Background and Motivation

> The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> *Edsger W. Dijkstra*

Isolation in computer systems keeps workloads apart so that the behavior of one workload does not affect other workloads. Perfectly isolating workloads in a cloud environment is a long-standing problem in systems [85, 65, 138, 69]. This chapter provides background on the virtualization technologies and isolation platforms this dissertation studies, and on the tension between isolation, performance, and efficiency that motivates our work.

2.1 Isolation Platforms

In this section, we describe the four isolation platforms that we will analyze (§2.1.1-2.1.4). First, we introduce LXC-based containers (§2.1.1), as implemented by Docker's default container engine, *runc*. Although LXC is sometimes used to refer specifically to the liblxc library [31] that creates and manages containers, we use LXC to more broadly refer to the "capabilities of the Linux kernel (specifically namespaces and control groups) which allow sandboxing" [28]; we explore these features as used by *runc*. Second, we describe KVM/QEMU (§2.1.2), a popular virtual machine platform. We then give an overview of

gVisor (§2.1.3) and Firecracker (§2.1.4), two platforms that both use a mix of LXC and KVM/QEMU components. gVisor's OCI-compliant [33] *runsc* engine and ongoing efforts to implement a similar engine for Firecracker [29] suggest it will soon be trivial to choose and switch between LXC, gVisor, and Firecracker when deploying with tools such as Docker and Kubernetes. We close with a brief overview of other platforms in the same design space (§2.1.5).

2.1.1 Linux Containers (LXC)

Linux Containers (LXC) [32, 10] are an OS-level virtualization method for running multiple isolated applications sharing an underlying Linux kernel. A *container* consists of one or more processes (generally running with reduced privileges) having shared visibility into kernel objects and a common share of host resources.

Shared visibility into kernel objects is governed by *namespaces*, which prevent processes in one container from interacting with kernel objects, such as files or processes, in another container. Resource allocation is governed by *cgroups* (control groups), provided by the kernel to limit and prioritize resource usage. An LXC container is a set of processes sharing the same collection of namespaces and cgroups.

Docker is an extension of LXC that adds a user-space *Docker daemon* to instantiate and manage containers. The Docker daemon needs root privileges, although a version that drops this requirement is under development. The daemon can be configured to launch containers with different container engines; by default, Docker uses the *runc* engine. It can also be configured to use other container engines such as *runsc*, which wraps gVisor (§2.1.3). Unless otherwise stated, any evaluation of "Docker" refers to an evaluation of "runc". The *runc* engine enables isolation for a running container by creating a set of namespaces and cgroups. Kubernetes functions like Docker in managing and launching containers.

Docker can also initialize storage and networking for the container engine to then use. The storage driver provides a union file system, using the Linux kernel, which allows

sharing of read-only files. Files created inside a Docker container are stored on a writable layer private to the container provided through a storage driver. Each container has its own virtual network interface. By default, all containers on a given Docker host communicate through bridge interfaces, which prevents a container from having privileged access to the sockets or interfaces of another container. Interactions between containers are possible through overlay networks or through containers' public ports.

2.1.2 KVM/QEMU

KVM (Kernel-based Virtual Machine) [22] is a type-2 hypervisor built into the Linux kernel that allows a host machine to run multiple, isolated virtual machines. QEMU [26] is a full-system emulation platform that uses KVM to execute guest code natively using hardware-assisted virtualization and provides memory management, device emulation, and I/O for guest virtual machines. KVM/QEMU provides each guest with private virtualized hardware, such as a network card and disk. On this platform, most operating system functionality resides in the guest OS running inside a virtual machine and in the QEMU process that performs memory management and I/O.

While commonly used for full-system virtualization and in infrastructure-as-a-service clouds, KVM/QEMU virtualization is heavyweight, due to both the large and full-featured QEMU process and to full OS installations running inside virtual machines. As a result, it is costly to run individual functions in a serverless environment.

2.1.3 gVisor

Google gVisor [18, 46] is a sandboxed container runtime that uses paravirtualization to isolate containerized applications from the host system without the heavy-weight resource allocation that comes with full virtual machines. It implements a user space kernel, *Sentry*, that is written in the Go Language and runs in a restricted seccomp container. Figure 2.1 shows gVisor's architecture. All syscalls made by the application are redirected into the

Sentry, which services them with its own implementations of kernel functionality–memory management and page-fault handling, threading, signal handling, and, as described below, file access and networking, covering the 237 syscalls it supports. Sentry makes calls to 53 host syscalls to support its operations. This prevents the application from having any direct interaction with the host through syscalls. gVisor supports two methods of redirecting syscalls: *ptrace-mode* uses ptrace in the Linux kernel to forward syscalls to the sentry and *KVM-mode* uses KVM to trap syscalls before they hit the Linux kernel so they can be forwarded to the sentry. As KVM-mode performs better than ptrace for many workloads [150] and has several benefits over the ptrace platform according to the gVisor documentation [82], we perform all experiments with it enabled.

gVisor starts a Gofer process with each container that provides the Sentry with access to file system resources. Thus, a compromised Sentry cannot directly read or write any files. A writable tmpfs can be overlaid on the entire file system to provide complete isolation from the host file system. To enable sharing between running containers and with the host, a shared file access mode may be used.

gVisor has its own user-space networking stack written in Go called *netstack*, to further isolate the sandbox from the host. The Sentry uses netstack to handle almost all networking, including TCP connection state, control messages, and packet assembly, rather than relying on kernel code that shares much more state across containers. gVisor also provides an option to use host networking for higher performance.

2.1.4 Firecracker

AWS Firecracker [16, 70, 108] is a virtual machine monitor (VMM) that uses KVM to create and run virtual machines. Like KVM/QEMU, it uses the KVM hypervisor to launch VM instances on a Linux host with a Linux guest operating system. It has a minimalist design and provides lightweight virtual machines termed *microVMs*. A *microVM* is a virtual machine with a minimal device model and stripped-down guest-facing function-

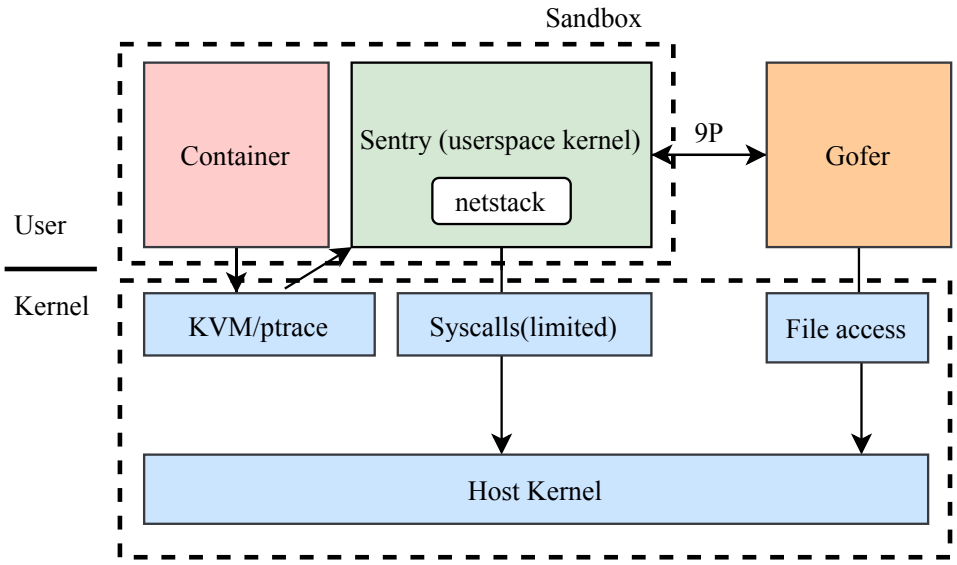


Figure 2.1: gVisor architecture

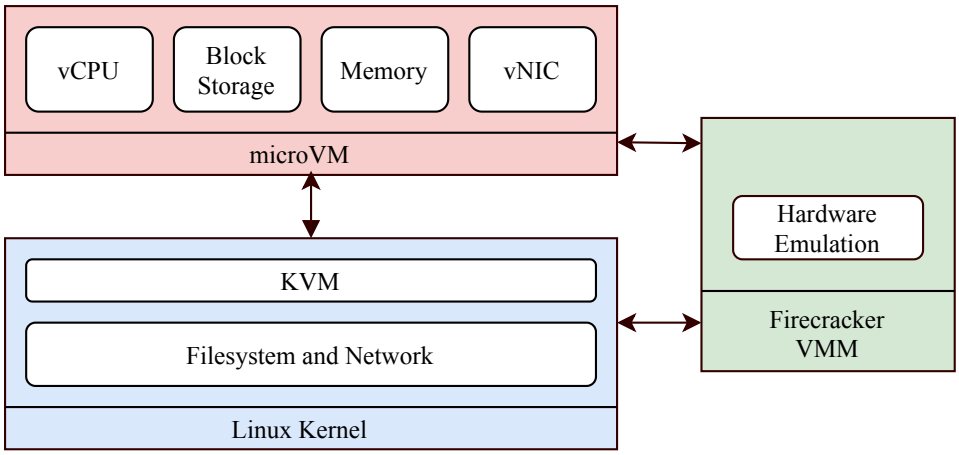


Figure 2.2: Firecracker architecture

ality. Firecracker excludes unnecessary devices and guest-facing functionality to reduce the memory footprint and attack surface of each microVM. Multiple microVMs can be launched by running separate instances of Firecracker, each running one VM. Compared to KVM/QEMU, Firecracker has a minimal device model to support the guest operating system and stripped-down functionality. Both of these allow it to make fewer syscalls and execute with fewer privileges than KVM/QEMU. Each *microVM* runs a full Linux guest kernel: Firecracker loads the guest kernel image directly, without a BIOS or bootloader,

and guests typically use a minimal kernel configuration.

As shown in Figure 2.2, each microVM runs as a process within the host OS, which is associated with a dedicated socket and API endpoint. Firecracker provides VirtIO/block and VirtIO/net emulated devices which are backed by files and TAP devices on the host respectively. To ensure fair usage of resource, it enforces rate limiters on each volume and network interface, which can be created and configured by administrators.

Each Firecracker process is started by a *jailer* process. The jailer sets up system resources that require elevated permissions (e.g., cgroups, namespaces, *etc.*), drops privileges, and then launches the Firecracker binary with `exec`, which then runs as an unprivileged process. To further limit the syscalls, Firecracker uses the kernel's seccomp filters to restrict the set of available operations.

Both Firecracker and gVisor rely on host kernel functionality. Firecracker provides a narrower interface to the kernel by starting guest VMs and providing full virtualization, whereas gVisor has a wider interface by being paravirtualized. Although Firecracker uses a minimal device model, its design can be viewed as more heavy-weight than gVisor. They both have low overhead in terms of memory footprint. Firecracker and gVisor are written in Rust and Golang respectively, both being type safe languages adding to their security models.

2.1.5 Other Platforms

While this dissertation focuses on Linux containers, gVisor, and Firecracker, several other isolation platforms occupy nearby points in the design space. Kata Containers [21] runs each container inside a lightweight virtual machine, pairing an OCI-compatible runtime with a hardware-virtualized boundary. Cloud Hypervisor [11] is a Rust-based VMM that, like Firecracker, provides a minimal device model for cloud workloads, but targets longer-running VMs with a broader feature set. Nabla containers [51] build on the idea of *unikernels as processes* [147], taking the library OS approach: applications link against a unikernel-style

runtime so that only a handful of host syscalls (7) are ever used. WebAssembly-based sandboxes [143, 129] and language-runtime isolates such as V8 [136] confine tenants within a single process, trading weaker boundaries for near-instant startup. These platforms differ in mechanism but share the strategy of the platforms we study: narrowing the interface to the shared host, and our measurement methodologies apply to them directly; we discuss them further in Chapter 7.

2.2 Balancing Isolation, Performance, and Efficiency

There is a fundamental tension between isolation and efficiency: more efficient platforms (*e.g.,* Linux processes, containers) tend to share more resources, while stronger isolation platforms (*e.g.,* full VMs, microVMs) share less. The cost of stronger isolation is paid in dedicated resources. A full VM reserves guest memory up front and carries its own kernel, page cache, and device state. Temporal isolation behaves the same way: dedicating cores to a tenant or scheduling with rigid CPU reservations prevents interference, but idles that capacity whenever the tenant does not use it.

The cost of sharing, conversely, is paid in interference. Sharing improves resource utilization, but it can leave resources unavailable to a workload when it needs them: a burst of allocations by one container can deplete the memory another depends on, and a neighbor's file-system activity can fill the shared page cache and journal, degrading performance even when every workload remains within its assigned resource limits. Moreover, sharing makes some shared resources more vulnerable to security risks (*e.g.,* unwanted access), compromising isolation. The degree of resource sharing directly impacts the level of isolation these platforms provide, as the likelihood of interference between workloads increases as more resources are shared, thereby reducing isolation. Nonetheless, sharing is integral in the cloud, as it improves resource efficiency.

There is a similar interplay between isolation and performance. Opting for more iso-

lation (*e.g.,* deploying a workload in a VM rather than a container) lowers performance because the stronger boundary adds overhead to the workload's execution path: system calls, and I/O must cross the virtualization layer, and booting a guest kernel takes longer. Platforms with less isolation avoid this interposition and start faster, but expose workloads to the interference described above. While platforms vary in the isolation they *provide*, workloads vary in the isolation they *need*: sensitive workloads (*e.g.,* payments) require strong isolation, whereas latency-critical workloads [139] need performance stability more than batch workloads do. This persistent tension makes it difficult to know which side to favor for a given workload: *should we use a stronger isolation platform and pay its cost in performance and efficiency, or share more and risk interference?*

2.3 Summary

This chapter introduced the isolation platforms we study – Linux containers, gVisor, and Firecracker – which take different approaches to a common strategy: narrowing the interface to the shared host kernel. It also examined the tension this strategy navigates: stronger isolation costs efficiency and performance, while greater sharing risks interference.

— 3 —

Blending Containers and Virtual Machines: A Study of Firecracker and gVisor

> Science progresses best when observations force us to alter our preconceptions.
>
> *Vera Rubin*

This chapter evaluates the architectures of lightweight isolation platforms based on how they use functionality in the host kernel. We compare the Linux kernel code footprint and performance of three isolation platforms: Linux containers, AWS Firecracker, and Google gVisor. We perform the first fine-grained code coverage analysis of these isolation platforms to determine how kernel services are used differently by each isolation platform: we compare *which* kernel code executes for different workloads as well as the *frequency* of invocation across platforms. For example, we investigate the impact of gVisor's user-mode networking stack on its usage of kernel networking code.

We also perform microbenchmarking of the three platforms compared to native Linux to measure the performance impact of the architectural choices. For CPU, networking, memory, and file system workloads, we measure the achievable performance on all three platforms.

Some highlights of our findings are:

- The code coverage analysis shows that despite moving much operating system functionality out of the kernel, both gVisor and Firecracker execute substantially more Linux kernel code than native Linux alone, and that much of the code executed is not in different functions, but rather conditional code within the same functions executed by native Linux.

- The frequency of invocation varies widely: gVisor, for example, handles most memory mapping operations internally, and hence invokes the kernel much less frequently than Linux containers.

- Our performance results show that neither gVisor nor Firecracker is the best of all workloads; Firecracker has high network latency while gVisor is slower for memory management and network streaming.

The rest of the chapter is organized as follows. First, we evaluate the system calls available from each platform and the amount of kernel code executed on each platform (§3.1). Next, we separately look at the code coverage and performance of CPU (§3.2), networking (§3.3), memory management (§3.4), and file access (§3.5). Finally, we summarize our findings and discuss the limitations of our work (§3.6), and conclude (§3.7).

3.1 Isolation Platform Comparison

A major distinction between isolation platforms is the location of operating system functionality: is it all handled by the host kernel, or is it moved out to user-space processes or guest kernels within a VM? We evaluate isolation platforms by comparing how they utilize Linux kernel functionality. We measure the *code coverage* of common operations: how much Linux kernel code executes in response to different workloads on each platform? This assists us in understanding how functionality is split between system components.

Furthermore, isolation can directly reduce performance: additional safety checks, layers of indirection such as namespaces, overlay file systems, and virtual networking all add extra code to system call and I/O paths. For multiple workloads, we evaluate the performance of isolation platforms in order to determine the runtime cost of isolation. As a baseline, we compare against native Linux processes with no special isolation. In this dissertation, we have not evaluated KVM/QEMU.

3.1.1 Methodology

We run all our experiments on a Cloudlab [12] xl170 machine, with a ten-core Intel E5-2640v4 running at 2.4 GHz, 64GB ECC Memory (4x 16 GB DDR4-2400 DIMMs), Intel DC S3520 480 GB 6G SATA SSD and 10Gbps NIC. We run on Ubuntu 18.04 (kernel v5.4.13). We performed all the experiments on gVisor release-20200127.0 version and Firecracker v0.19.1. Since these systems are under continuous development, results might vary with older and future versions.

We run all experiments across four configurations: host Linux with no special isolation, Linux containers (labeled LXC), Firecracker, and gVisor. For gVisor, we use KVM-mode to trap system calls from a container. The official performance guide of gVisor [30] uses ptrace mode in almost all their experiments, so many of our performance results are different. For starting a Firecracker microVM, we set the memory to 8GB, storage to 4GB, and vCPU to 1. For gVisor and LXC, memory was set to 8GB by passing the –memory flag with *docker run*. Each platform's default file systems are used for the measurements.

3.1.2 Security Policies

The Linux kernel provides over 350 syscalls as entry points to operating system functionality. Each syscall is a possible vector for attack against the kernel, as a flaw in the kernel could allow user-mode code to subvert OS protection and security mechanisms. However, most

Platform	**Total allowed syscalls to the host kernel**
LXC	all except 44
Firecracker	36
gVisor w/o host networking	53
gVisor w/ host networking	68

Table 3.1: Total number of system calls allowed out of 350

programs use only a subset of the available system calls. The remaining system calls should never occur. If they do occur, it is perhaps because of a compromised program.

Linux provides a secure computing mode—*seccomp* [24]— to restrict the system calls a running process may make. We focus on use of *seccomp-bpf*, which allows a user space-created policy to define (a) the permitted syscalls, (b) allowed arguments for those syscalls, and (c) the action to be taken when an illegal syscall is made.

All three systems we study (LXC, gVisor, and Firecracker) use seccomp filters to reduce application-host kernel interaction, thereby increasing isolation and reducing the attack surface. A summary of the allowed syscalls in each system is shown in Table 3.1.

LXC can use seccomp [15] to limit actions available to the process running inside the container. By default, Docker blocks 44 system calls that are obsolete, can only be called with root privilege, or are not protected by kernel namespaces.

Firecracker has a whitelist of 36 syscalls [17] that it requires to function. It supports two levels of seccomp filtering. With *simple* filtering, only the 36 calls are permitted and the remainder denied; with *advanced* filtering, Firecracker adds constraints on the values allowed as arguments to system calls as well. A syscall is blocked if the argument values do not match the filtering rule.

gVisor limits the syscalls to the host by implementing most of them in the Sentry process [131]. The Sentry itself is only allowed to make a limited set of syscalls to the host. As of November 2019, Sentry supports 237 syscalls out of 350 within the 5.3.11 version of the Linux kernel. It uses a seccomp whitelist of 68 syscalls to the host with host networking enabled and 53 without networking.

	Host	Firecracker	LXC	gVisor
Lines	63,163	77,392	90,595	91,161
Coverage	7.83%	9.59%	11.23%	11.31%

Table 3.2: Union of line coverage across all workloads out of 806,318 total lines in the Linux kernel.

Overall, both gVisor and Firecracker provide much stronger system call limits than Linux containers, with Firecracker being a bit stricter due to its use of fewer system calls and tighter controls over the arguments to those system calls. In contrast, Linux containers allow isolated applications to make most system calls.

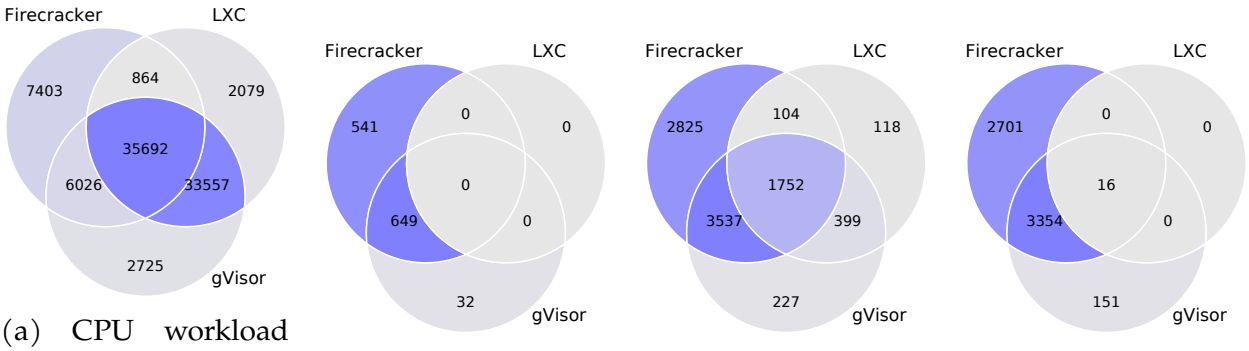
3.1.3 Total Code Footprint

We measure the code coverage for four different microbenchmarks, described in later sections: CPU, network, memory, and file write on the four systems (host, LXC, Firecracker and gVisor).

Each workload runs for ten minutes. We run the lcov [23] code coverage tool, version 1.14 on kernel v5.4.13, which reports which lines of source code were executed and what fraction of functions, lines, and branches of total kernel code were executed. It also reports how many times each line of code was executed. Using these results, we can compare the kernel code coverage for the same workload across different isolation platforms to see first, how much code they execute, and second, whether a platform uses the same code as another platform or does something different.

Table 3.2 shows the union of line coverage across the workloads. The lines row shows the number of lines of code executed at least once out of the total 806,318 lines of code in the Linux kernel. Overall, we find that native Linux executes the least code and that both Firecracker and gVisor, despite having a separate guest kernel and a user-space kernel respectively, execute substantially more Linux kernel code.

While this gives us a broad picture, it does not tell us anything about whether the platforms are largely executing the same code and more, or if there are large non-overlapping



(a) CPU workload overall (b) /virt (c) /arch (d) /arch/x86/kvm

Figure 3.1: CPU workload coverage

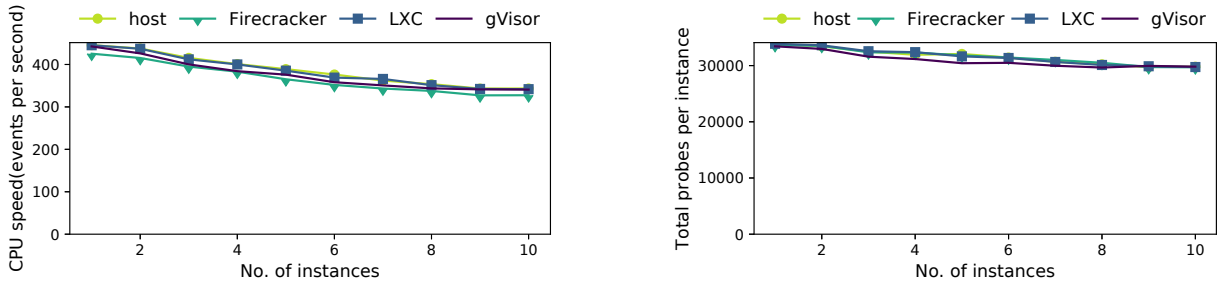
bodies of code. In the following sections, we take a detailed look at the code executed for different kernel subsystems when running microbenchmarks exercising that subsystem. Figures 3.1, 3.3, 3.10, and 3.15 show the intersection of code executed by each system, represented as a set of Venn diagrams.

When determining whether to consider a line of code as executed by a platform, we filter out invocations not related to the application running in the isolation platform. To do this, we measure the code coverage of an idle host Linux system for 10 minutes and calculate the *hit rate* of every line of code—how often it is called. When determining which code was executed by a platform, we compare the hit rates of the code with the platform against the idle system and only include the code if it was executed more frequently with the isolation platform present. With this mechanism, we minimize the noise from the background processes, though we cannot eliminate it completely in our measurements. Stripping idle code removes approximately 10k-15k lines of code, depending on platform.

3.2 CPU Workload

3.2.1 Coverage Analysis

We run the sysbench CPU benchmark [27] on each virtualization platform and identify the footprint in each case. Figure 3.1a shows the lines of code in each footprint across the entire



(a) CPU Speed

Figure 3.2: CPU workload

(b) LLC Probe

Linux kernel, and how the footprints overlap: darker portions of the diagrams correspond to larger line counts.

We see a majority of the lines (35,692) are shared by all three platforms; Firecracker has the most unique lines (7,403) not shared by any other platform, and LXC and gVisor share a significant footprint not exercised by Firecracker (33,557 lines). We note that although gVisor implements many system calls in the Sentry, it exercises more host code than LXC; although surprising, the same was found in prior analysis [51]. Overall, gVisor has the largest total footprint (78k LOC) and Firecracker has the smallest (49k LOC).

We illustrate the source of the differences between platforms by analyzing code in specific Linux source directories. The code in the */virt* directory shown in Figure 3.1b is executed only in Firecracker and gVisor, because they virtualize with KVM. Firecracker has the highest footprint here of 1,190 LOC (541+649) as it uses VirtIO emulated Network and Block devices, while gVisor relies on system calls for I/O, resulting in a footprint that is 43% smaller than and nearly a subset of Firecracker's footprint.

Firecracker and gVisor have a much larger */arch* footprint than LXC (Figure 3.1c); most of the additional code is under */arch/x86/kvm*, as shown in Figure 3.1d. Here, we see gVisor's footprint is 42% smaller than and nearly a subset of Firecracker's exercised lines of code. KVM code is used by Firecracker for microVMs and by gVisor to redirect system calls to the Sentry; however, Firecracker has 2,550 more lines of code than gVisor. This difference is mostly from the emulation code that Firecracker executes but gVisor does not.

Despite these substantial differences in functionality, much of the new code executed by gVisor and Firecracker is not in new functions. Rather, it is conditional code in the same functions exercised by containers. For instance, most of the code in the file *block/blk-cgroup.c* is executed just by gVisor as an extra check. We will see more such examples in the following sections.

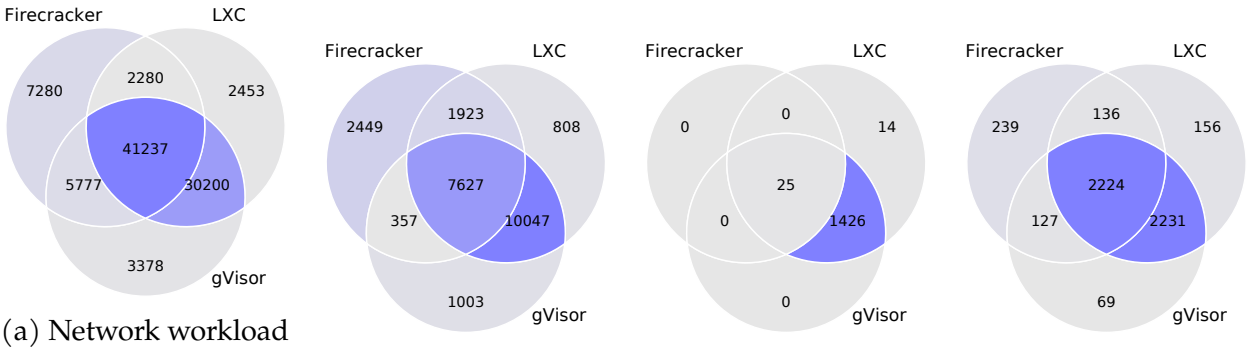
3.2.2 Performance

We conduct two experiments to measure how the choice of isolation platform affects CPU performance. First, we run the sysbench CPU benchmark [27] for ten seconds and observe the number of events it executes. An event is the loop that finds prime numbers up to a limit. This workload is CPU-bound, and hence measures the processing overhead of the platforms. We observe in Figure 3.2a that all the platforms perform similarly. As we increase the number of instances to ten, performance per instance drops 23%.

In the second experiment, we execute LLCProbe [137], which sequentially probes 4B from every 64B of data within an LLC-sized buffer, using cache coloring to balance access across cache sets. To measure the performance, we measure the number of probes completed in 10 seconds. Figure 3.2b reports the average probes observed per instance. There is a slight drop in performance as the number of instances increases across all environments, due to the overhead of the platform. All platforms achieve approximately 33k probes per instance.

3.2.3 Insights

Despite the user-mode Sentry, *gVisor has the largest footprint and Firecracker the smallest*. But all three platforms exercise a significant body of code not exercised by the other two. While not all lines of code are equally likely to be vulnerable to an exploit, more code indicates potentially higher risk.



(a) Network workload overall (b) /net (c) /net/bridge (d) /net/core

Figure 3.3: Network Workload Coverage

Firecracker and gVisor require more architecture-specific code due to their dependence on hardware virtualization, which may make portability to other processor architectures more difficult. Firecracker depends on KVM to achieve the lightweight virtualization, adding to the lines of code it executes in the kernel. gVisor is heavily paravirtualized and also uses a smaller fraction of this functionality when running with the KVM platform. LXC does not use this functionality as it lies on the leftmost side of the isolation spectrum as seen in Figure 1.2.

The CPU performance corresponding to these architectural differences does not vary significantly for both workloads, as overall there is fairly little kernel involvement.

3.3 Networking

3.3.1 Coverage Analysis

We measure code coverage of network operations for each platform using iperf3 [20], which streams data over TCP/IP. Figure 3.3a shows the overall kernel footprints and Figure 3.3b shows the footprints in the */net* directory alone. Despite having an in-Sentry networking stack (including TCP, UDP, IP4, IP6 and ICMP), gVisor has the highest overall coverage and surprisingly exercises much of the same code as LXC under */net*. Some parsing of packets still happens in the host kernel as a sanity check before routing them. Similarly,

```
bool is_skb_forwardable(const struct net_device *dev, const struct sk_buff *skb){
    unsigned int len;
    if (!(dev->flags & IFF_UP))
        return false;
    len = dev->mtu + dev->hard_header_len + VLAN_HLEN;
    if (skb->len <= len)
        return true;
    if (skb_is_gso(skb))
        return true;
    return false;
}
```

(a) net/core/dev.c

	Hits		
Lines	**LXC**	**gVisor**	**Firecracker**
1823	1.012e+8	1.805e+6	0
1825	2.111e+8	5.099e+6	0
1827	2.111e+8	5.099e+6	0
1828	2.111e+8	5.099e+6	0
1830	8.270e+6	9.974e+5	0

(b) Number of hits in net/core/dev.c

Figure 3.4: Hits in net/core/dev.c

when Generic Segmentation Offload (GSO) is enabled in gVisor, the host kernel does more processing of packets. The */net/bridge* and */net/core* directories also show a high overlap among them, with 1,426 and 2,231 lines shared by LXC and gVisor as presented in Figures 3.3c and 3.3d respectively.

Firecracker runs the fewest lines of networking code. This reflects that most networking happens in the guest OS.

3.3.2 Code Differences

In comparison to the host, all three isolation platforms execute more setup code for their network interfaces and more extra checks across all networking code. *dev_get_by_index*(), defined in */net/core/dev.c*, searches for a network device; it only gets executed in LXC and gVisor but not on the host and Firecracker, which suggests that gVisor and LXC have the

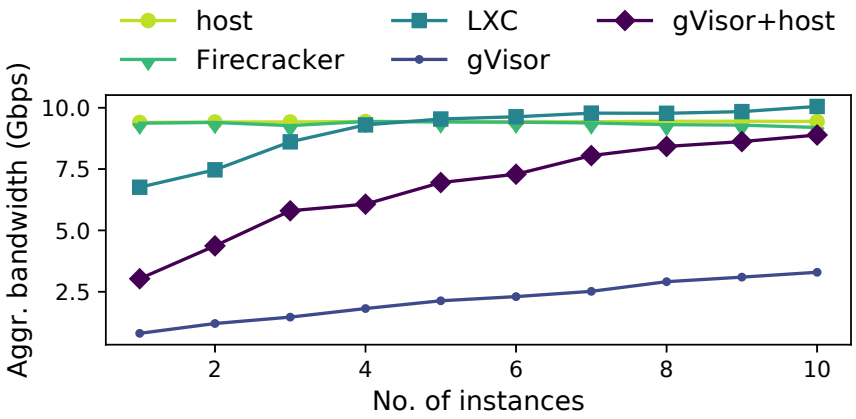


Figure 3.5: Aggregate Network Bandwidth

same type of network interface, i.e., a bridge network.

There are some functions that execute only on some of the platforms, such as *tcp_sum_lost* in */net/ipv4/tcp_input.c*, which checks the sum of the packets lost on the wire. This gets executed 27,719 times in LXC, 3 times in Firecracker, and is not executed in gVisor. This shows functionality that gVisor handles in Sentry; hence, not using the kernel network stack for this processing.

Figure 3.4a shows the code snippet and Figure 3.4b shows the hits (number of times a line executes) for some of the lines. LXC has the highest hits, more than 100 million, followed by gVisor, which calls this function 1,805,020[12] times showing that it handles part of this inside netstack. Firecracker does not execute this code.

3.3.3 Performance

We evaluate network performance by measuring bandwidth and latency. The network bandwidth is measured by running iperf3 between two Cloudlab machines with a 10 Gbps network link, and the latency by measuring the round-trip time with ping.

[1]https://stackoverflow.com/questions/46447674/function-coverage-is-lesser-even-with-100-code-coverage-when-objects-created-in
[2]https://stackoverflow.com/questions/2780950/gcov-line-count-is-different-from-no-of-lines-in-source-code

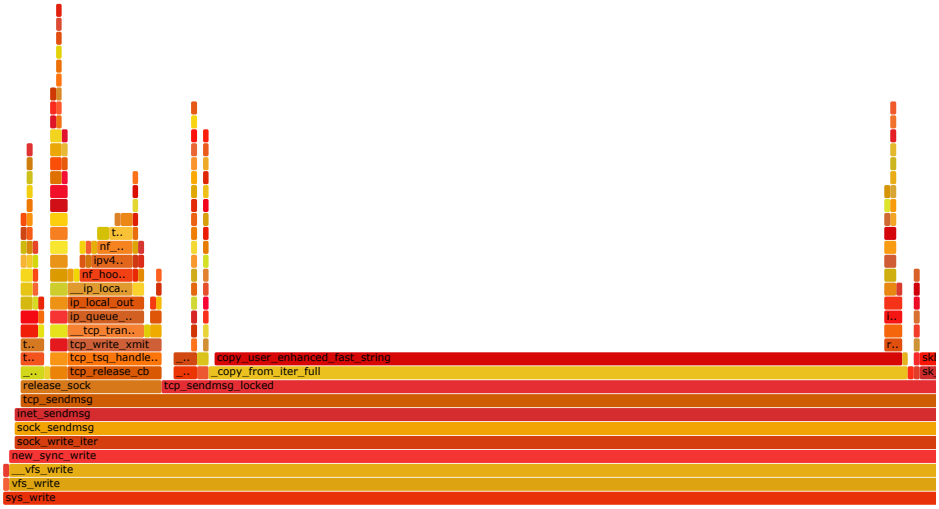


Figure 3.6: Host network profile

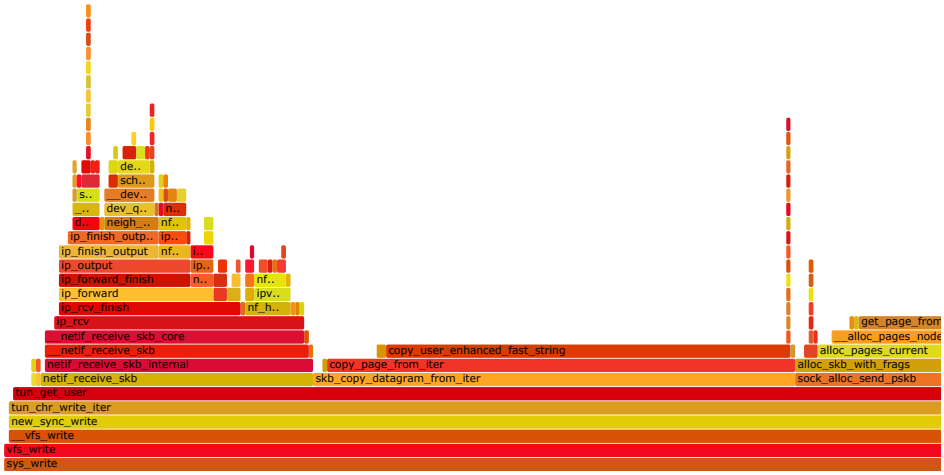


Figure 3.7: Firecracker network profile

Host and Firecracker achieve the highest bandwidth; LXC is slightly slower, while gVisor is the slowest with 0.805 Gbps[3] (805 Mbps) but the aggregate increases to 3.294 Gbps at 10 instances. This is likely due to its user space network stack, which is not as optimized as the Linux network stack. When using the host networking stack, gVisor is still slowest, but achieves 3.03 Gbps. The performance of the ptrace platform, as stated in the gVisor official guide [30] is faster. The gVisor network performance for the KVM platform has increased substantially from its release-20190304 version, by nearly 800%.

To better understand how the different platform architectures use the kernel differently,

[3]We get similar results on Google Cloud Platform (GCP).

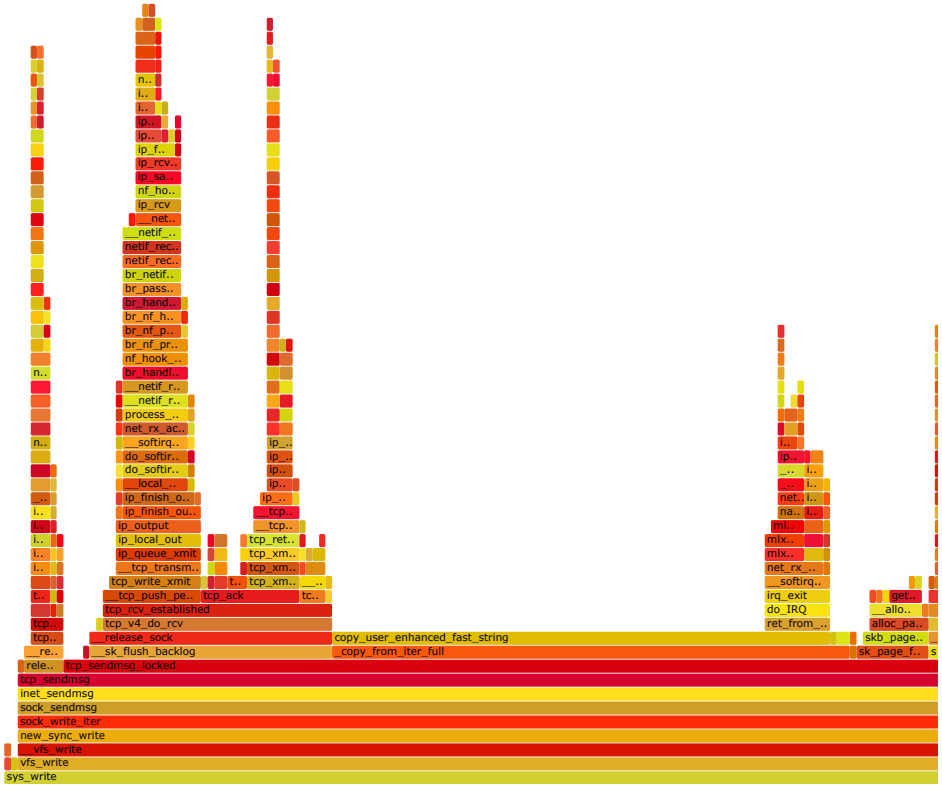


Figure 3.8: LXC network profile

we show flame graphs [79] for the *sys_write* system call in Figures 3.6–3.9. Each rectangle represents a function, and the stacks represent call stacks. The width represents the fraction of time that function was on the call stack. Several patterns are visible. Compared to host networking in Figure 3.6, LXC in Figure 3.8 has a much taller stack, which represents the cost of traversing two network interfaces to implement bridging. For gVisor, the trace is collected for the Sentry process that does a lot of processing on the packet. Firecracker runs TCP in the guest, so it invokes packet forwarding code at the IP layer and has a much flatter profile.

We show the aggregate bandwidth as the number of instances increases in Figure 3.5. With increasing instances, there is almost no change in aggregate and average bandwidth in most cases. For gVisor with host networking, the aggregate increases with more instances, but peaks below the other platforms at 10 instances.

We measured the round-trip time (RTT) by running ping for ten seconds and taking

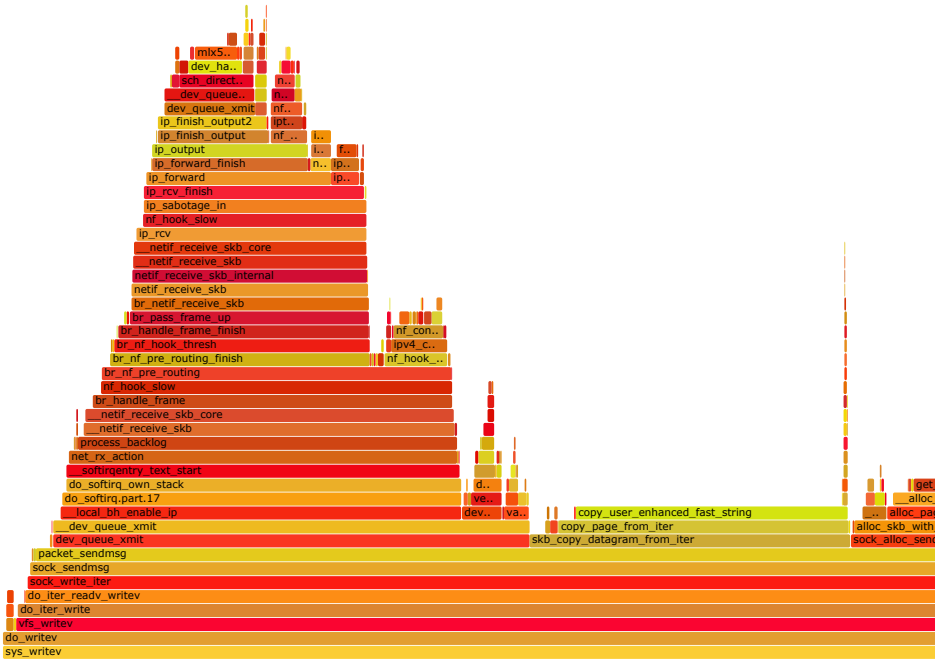


Figure 3.9: gVisor network profile

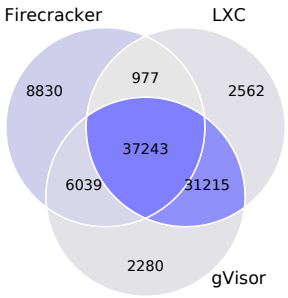
	Host	Firecracker	LXC	gVisor
RTT (µs)	146	371	149	319

Table 3.3: Round-trip time

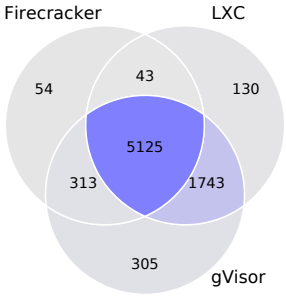
the average RTT shown in Table 3.3. Firecracker incurs maximum latency, followed by gVisor, LXC, and the host. In the case of Firecracker, a packet traverses through two levels of the kernel network stack, which explains its high latency. gVisor does not go through all the layers of the host network, making its latency slightly better than Firecracker, but still far from host or LXC.

3.3.4 Insights

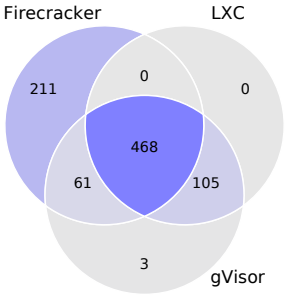
For the network workload we observe that *Firecracker has most of the implementation inside the guest and uses less functionality of the host compared to gVisor and LXC*. It also shows high performance. *Even though gVisor has its own networking stack, it touches a lot of kernel code*. LXC uses much of the same code as the host, but adds substantially to it for bridging, which comes at little performance cost.



(a) Memory workload overall



(b) /mm



(c) /arch/x86/mm

Figure 3.10: Memory workload coverage

Although LXC has high performance, it relies completely on the kernel for all the packet processing and sanity checks, which lowers its isolation. gVisor does most of the processing inside netstack, but some checks are still performed at the host, which provides more isolation. Firecracker has almost all the processing inside the guest, which makes its networking stack relatively more isolated.

3.4 Memory management

3.4.1 Coverage Analysis

We show the overall kernel coverage in Figure 3.10a for a workload that mmaps and munmaps regions. The */mm* directory in Figure 3.10b shows 1,743 lines of common code between LXC and gVisor. Memory management in gVisor [19] has two levels of virtual to physical page mapping, one from application to Sentry and the other from Sentry to host. Sentry requests memory from the host in 16MB chunks to minimize the number of mmap() calls. For a 1GB memory request by an application, Sentry will make 64 mmap() calls to the host. Although gVisor reduces the number of such calls, it still executes the same lines of code as LXC for any memory request. Thus, we observe both having similar coverage in this directory. Firecracker has lower coverage here with only 54 unique lines of code. It does not make any mmap() calls to the host after the first initialization needed to

run the microVM and has only 54 unique lines of code.

Figure 3.10c shows the architecture-specific memory management code, with Firecracker having 211 unique lines of code. LXC and gVisor have similar footprints in this directory. Although gVisor implements memory management in the Sentry, it still depends on the host for this functionality like LXC, whereas Firecracker implements the whole memory stack inside the guest and does not use this after running some initial architecture-specific setup code.

3.4.2 Code Differences

Several kernel functions show significant differences in invocation frequency. The *do_mmap*() function in *mm/mmap.c* for the memory workload is called more than 1 million times by LXC, but only 5,382 times from gVisor, due to the two-level page table implemented in Sentry that reduces the number of syscalls to the host. Firecracker invokes this function only 3,741 times, as it is only used by the VMM for physical page management and not in response to guest mmap() calls.

zap_page_range_single() defined in *mm/memory.c* which removes user pages in a given range is executed 459,984 times in gVisor and 0 times in Firecracker and LXC. This is likely due to the memory management implementation inside the Sentry process.

The *vma_wants_writenotify*() in */mm/mmap.c* checks for write events in pages marked read-only. This function gets called in all three platforms, but only Firecracker and gVisor execute the conditional code inside it.

3.4.3 Performance

We benchmark memory allocation cost by allocating and freeing memory with mmap() and munmap() system calls. The benchmark allocates 1 GB of memory using mmap() calls with varying granularity from 4KB to 1MB, so with larger requests there are far fewer allocations. After each call, the test program touches one word of each allocated page by

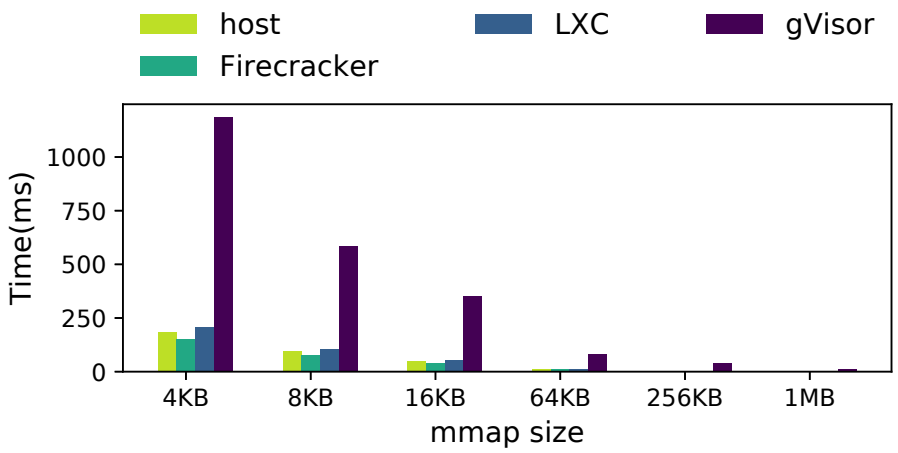


Figure 3.11: Total allocation time (without munmap) for 1GB

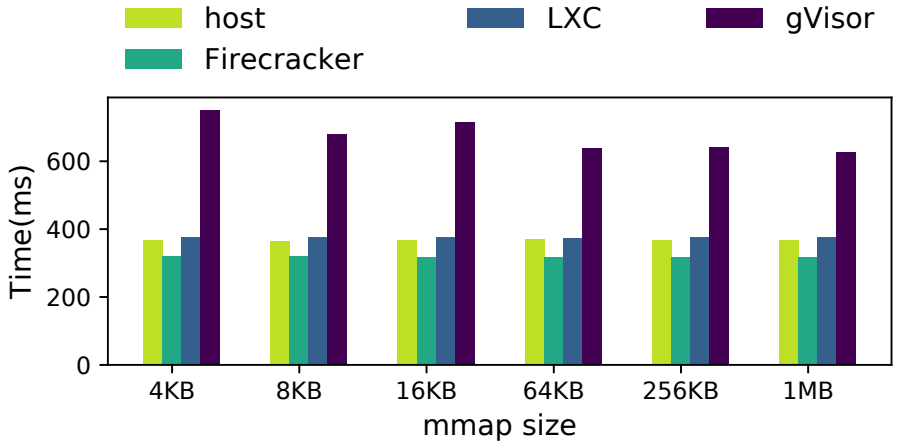


Figure 3.12: Total touch time (without munmap) for 1GB

writing to it. We also run the same experiment with calling munmap() after touching the pages of each allocation so physical memory and virtual addresses can be recycled.

In Figures 3.11 and 3.12, we present the time taken for mmap() and touch, respectively. In this case, we do not unmap pages after each allocation. In Figure 3.13, time includes mmap() and munmap(), and the touch time is presented separately in Figure 3.14.

For both cases, gVisor is expensive, taking the highest time for allocation, allocation+munmap, and touch. However, we observe that it becomes competitive with the increase in mmap size. We suspect this is due to the two levels of page mapping, from

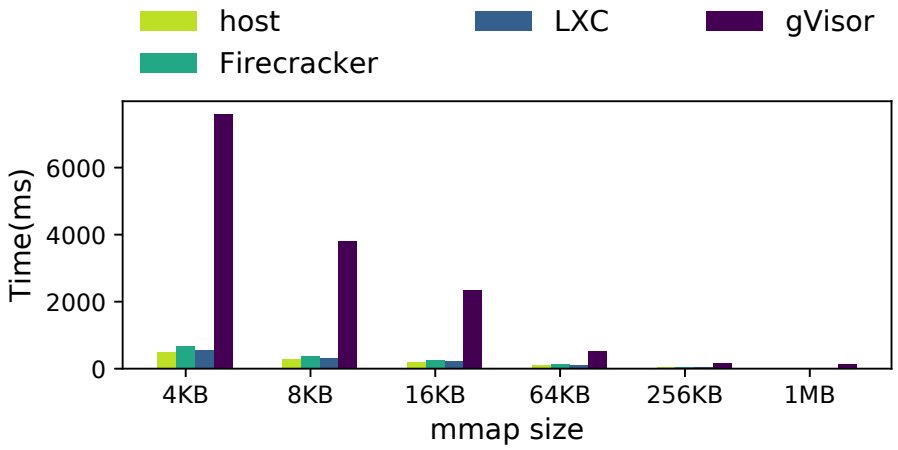


Figure 3.13: Total allocation+unmap time for 1GB

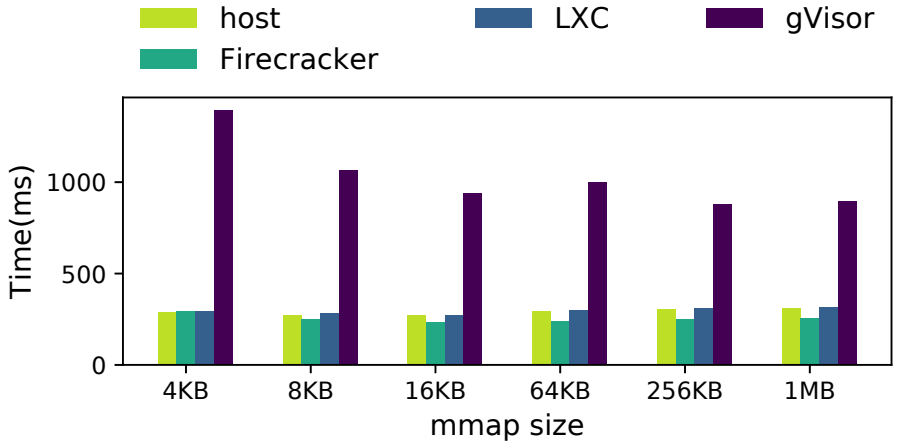
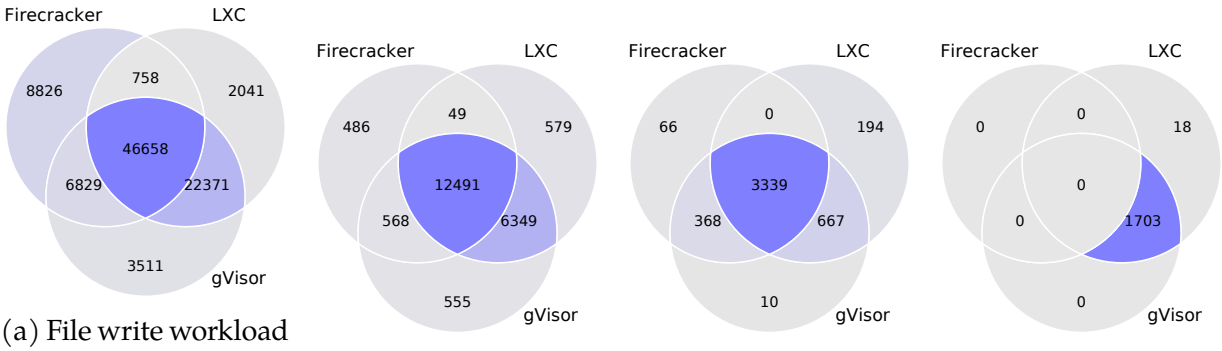


Figure 3.14: Total touch time (with munmap) for 1GB

application to Sentry and then to the host, which requires management of more code and data structures contributing to this overhead. Firecracker and LXC perform similarly to the host in most of the cases, but become more expensive if we unmap, likely due to higher costs of shootdowns on virtualized platforms.

3.4.4 Insights

The large difference in invocation frequency for gVisor occurs because it has moved some of the memory management inside Sentry. This design makes it very expensive for workloads that



(a) File write workload overall (b) /fs (c) /fs/ext4 (d) fs/overlayfs

Figure 3.15: File write workload coverage

do allocation in smaller chunk sizes. *Firecracker does not use any of the host functionality after initializing the microVM*, making the memory stack fully isolated with high performance in most cases. But system virtualization becomes slower when we start unmapping and re-mapping, so for workloads that do a lot of mmap() and munmap(), this might come with an overhead. *LXC heavily uses the kernel code with very high hit rates*, which makes it less isolated. But it has high performance, better than Firecracker in some cases.

3.5 File access

3.5.1 Coverage Analysis

We measure coverage with a workload that opens, reads, and writes to files. Figure 3.15a shows the overall kernel footprints and Figure 3.15b shows the footprints in the */fs* directory, which includes VFS code as well as the implementations for various file systems. We observe that most lines (about 18k) are shared between gVisor and LXC; Firecracker uses about half of these lines as well.

In our experiments, all three systems use the ext4 file system. Figure 3.15c shows that gVisor and LXC exercise more ext4 code than Firecracker. Storage in Firecracker is provided by an emulated block device, which is backed by a file on the host, so there are no directory operations.

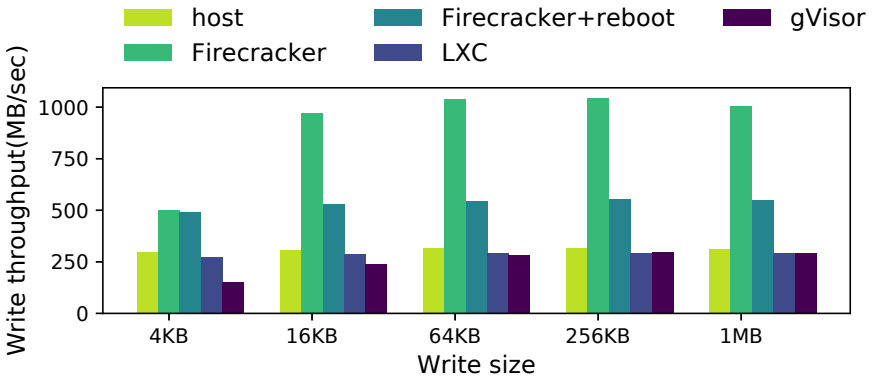


Figure 3.16: Write Throughput

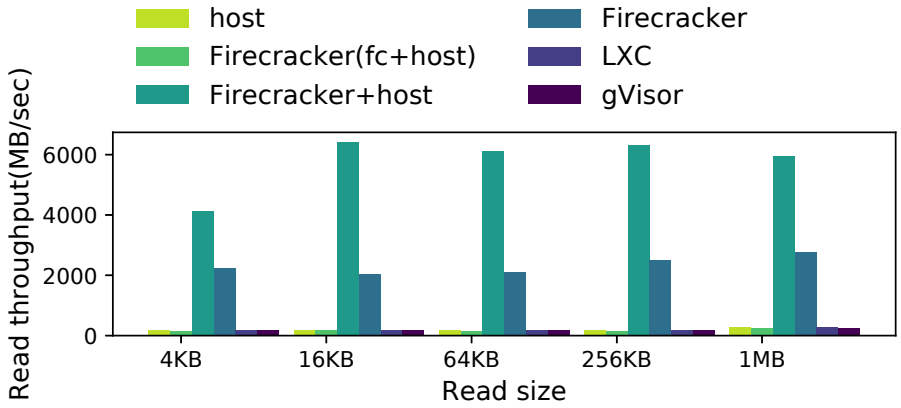


Figure 3.17: Read Throughput

gVisor and LXC containers do not directly access ext4 files; rather, they interact with overlayfs, which provides copy-on-write and stitches together a single file-system view from multiple ext4 directories. Figure 3.15d shows that LXC and gVisor have a similar overlayfs footprint, and that (as expected) Firecracker makes no use of overlayfs.

3.5.2 Performance

We measure I/O throughput with a microbenchmark that performs reads and writes of various sizes on a 1 GB file. For writes, the test creates the file and then writes sequentially. After each iteration, the file is flushed to disk; the cache is cleared between each read test.

For the write throughput shown in Figure 3.16, we run two variations for Firecracker.

In the first case, we run the test repeatedly on the same microVM. In the second case, we reboot the microVM before each test. This causes Firecracker to close and reopen the backing file for its block device, which may account for the lower performance following a reboot. Nonetheless, in both cases Firecracker is much faster than the other platforms because it makes no effort to write data back to persistent storage. As a result, it operates much closer to the speed of memory. In contrast, gVisor and LXC perform very similarly to native Linux; gVisor is a bit slower for small files where the overhead of calling through the Sentry is not masked by data movement costs.

Read throughput is shown in Figure 3.17. Host, LXC, and gVisor all perform similarly across all read sizes, as they all must read from storage. For Firecracker, we run three variations: plain *Firecracker* flushes the file cache in the microVM only; *Firecracker+host* flushes the cache only in host Linux and not in the microVM, and *Firecracker(fc+host)* flushes caches in both locations. When all caches are flushed, Firecracker behaves similarly to the other platforms, as it must fetch data from storage. When only the cache in the microVM is flushed, performance is closer to memory speed, but still pays the cost of exiting the VM to copy data from the cache in the host kernel. Finally, when just the host cache is flushed, performance is near memory speed: the data can still be accessed from the guest OS cache. Overall, a warm cache within the microVM results in nearly double the throughput of a warm cache in the host.

3.5.3 Insights

Firecracker only uses ext4 as a backing for a virtual disk; this results in only a moderate footprint since this use case only involves file-system data, not metadata. In contrast, *gVisor and LXC use overlayfs over ext4; this results in an increased footprint for overlayfs, as well as more ext4 code being exercised* (overlay file systems heavily use metadata in the underlying file system).

3.6 Summary

Outcomes The preceding sections analyzed how LXC, Firecracker, and gVisor make different use of kernel functionality, and how they exercise different kernel code. Our analysis demonstrated several aspects of these isolation platforms.

First, using Linux containers, while highly performant, greatly increases the amount of kernel code exercised. In some cases, this was new functionality that greatly lengthened code paths, such as networking where bridging expanded the amount of code needed for transmitting data. However, in many cases the new code was not an entire module, but instead was conditional code interspersed with code run by native Linux. In such cases, existing functions execute more slowly due to the added code.

Second, Firecracker's microVMs are effective at reducing the frequency of kernel code invocations, but have a much smaller impact on reducing the footprint of kernel code. Indeed, running workloads under Firecracker often expanded the amount of kernel code executed with support for virtualization. In addition, Firecracker's use of more limited functionality, such as memory mapping only at startup, file I/O for data but not directories, and networking at the IP level, show that future kernel releases targeting microVMs could aggressively optimize specific hot code paths at the expense of sacrificing performance of more general workloads needed by a general purpose OS.

Finally, we found that the gVisor design leads to much duplicated functionality: while gVisor handles the majority of system calls in its user-space Sentry, it still depends on the same kernel functionality as Linux containers. Thus, its design is inherently more complicated, as it depends on multiple independent implementations of similar functionality. In contrast, LXC relies on a single implementation of OS functions in the Linux kernel, and Firecracker relies on a much-reduced set of kernel functionality for performance.

We believe our findings serve as an architectural guide for building future systems and also be a motivation for future research concerning the coverage and security of new isolation platforms.

Limitations. Our methodology is a middle ground between full-system profiling and foreground-only profiling, and may err by capturing a bit of background activity, even though the goal is capturing all kernel activity induced from the isolation platform. Furthermore, while we focus on code coverage, studying coverage with microbenchmarks only provides a hint as to the security of a system; more investigation of coverage under richer workloads, including exploring the total coverage available from a platform, may be needed to make stronger security claims.

The microbenchmarks we use for the study allow us to stress particular subsystems and study their performance and coverage. While running more complex workloads or test suites would provide more information about total code coverage, their complexity could obscure significant differences in usage of different subsystems. Moreover, running existing containerized benchmarks is a challenge because Firecracker is not yet integrated with Docker.

3.7 Conclusion

In this chapter, we presented a system-wide Linux code-coverage analysis for three isolation platforms. This enables us to study the impact of different isolation architectures that move more and more code out of the kernel. We also ran performance benchmarks to understand the runtime cost of each platform. Our results show that despite adding substantial code outside the kernel for OS functionality, both gVisor and Firecracker still exercise more kernel code for most workloads.

— 4 —

The Kernel Always Knows: Understanding Isolation via Kernel Objects

> Good fences make good neighbors.
>
> *Robert Frost, Mending Wall*

This chapter analyzes isolation at the granularity of kernel objects: the kernel-resident data structures (*e.g.*, inodes, page allocators, scheduling queues) through which even highly isolated platforms ultimately share the host kernel. Resource-partitioning mechanisms control how much of a resource a workload may consume, but they do not eliminate interference caused by shared access to these objects, and existing profiling tools expose where time is spent rather than which objects are shared. We leverage the observation that the operating system, by design, synchronizes access to shared kernel objects, and use synchronization events as a proxy for potential object interference between workloads.

We present LIMA (Lock Interference Mapping Analyzer), a tool that combines dynamic tracing of kernel locking events with static analysis to map kernel locks to the underlying data structures they protect. We use LIMA to conduct a comparative study of the sharing and interference observed at the system software level when concurrent workloads execute on Linux containers, gVisor, and Firecracker. Our focus is on the isolation of system *structure*, meaning how well isolated the data structures of system software are; hardware isolation issues that are orthogonal to software structure (*e.g.*, micro-architectural side-channels)

are out of scope [42, 99, 98, 110].

Some highlights of our findings are:

- Object-level isolation is not a static property of the platform but a dynamic function of the workload: workloads that rarely synchronize operate on effectively disjoint objects regardless of platform, while system-service-heavy workloads expose substantial shared state.
- Isolation is shaped by platform design: Linux containers share a wide variety of objects with high intensity; Firecracker narrows this variety but accesses specific hot objects intensely, especially during initialization; and gVisor shares across a moderate number of objects while maintaining lower acquisition rates.
- Memory allocation and file-system journaling are the dominant sources of cross-application interference across our collected traces.

The rest of this chapter is organized as follows. We first provide background on kernel objects and motivate the need for an object-level view of isolation (§4.1), and present a case study of object-level interference (§4.2). We then describe the design of LIMA (§4.3) and our object analysis methodology (§4.4), present our object-level interference analysis across workloads and platforms (§4.5), discuss the implications of our findings (§4.6) and conclude (§4.7).

4.1 Background and Motivation

4.1.1 Isolation and Interference

Isolation in computer systems keeps workloads apart from each other, so that the behavior (functional or temporal) of one workload does not affect other workloads. The goal for perfect isolation is *non-interference* [111, 96], where a workload executes identically to how

Platform / Resource	KVM	LXC	gVisor	Firecracker
Kernel	Isolated	Shared	Shared	Isolated
Network	Same as host	Same as host	Shared/ Isolated	Same as host
Memory Allocation	Isolated	Shared	Shared/ Isolated	Isolated
Filesystem	Isolated	Shared/ Isolated	Shared/ Isolated	Isolated
Hardware	Shared	Shared	Shared	Shared

Table 4.1: Sharing across different platforms.

```
1  void set_fs_pwd(struct
       fs_struct *fs, struct path *
       path){
2    struct path old_pwd;
3    path_get(path);
4    spin_lock(&fs->lock);
5    // sets fs->pwd to *path
6    spin_unlock(&fs->lock);
7  }
```

(a) Private

```
1  void release_task(struct
       task_struct *p){
2    write_lock(&tasklist_lock);
3    // removes task from list
4    write_unlock(&tasklist_lock);
5  }
```

(b) Shared

```
1  void insert_inode_hash(struct
       inode *i, u64 hval){
2    struct hlist_head *b =
3      inode_hashtbl + hash(i->
         i_sb, hval);
4    spin_lock(&inode_hash_lock);
5    hlist_add_head_rcu(&i->
         i_hash, b);
6    spin_unlock(&
         inode_hash_lock);
7  }
```

(c) Incidentally shared

Figure 4.1: Examples of lock usage patterns in the Linux kernel.

it would when run alone. While often used for security by looking at what is possible for an attacker, for isolation, we consider it in terms of *structural independence*: a workload should function as if it has exclusive access to the system resources and kernel state required for its execution. Thus, an imperfect isolation mechanism can provide perfect isolation if the two workloads do not exercise or contend on shared system facilities that allow interference.

The basic approach that isolation systems take is to separate workloads in space and time. Spatial isolation ensures workloads access different data or hardware (*e.g.*, separate memory pages, dedicated CPU cores), while temporal isolation prevents them from accessing the same data or hardware simultaneously (*e.g.*, CPU time-slicing). These approaches are highly effective for resource management, as they control the amount of resources a workload receives. Mechanisms like Linux cgroups [90], qdisc [86], and cache parti-

tioning [53] enforce strict spatial and temporal boundaries on resource allocation when sufficient resources are available.

However, resource partitioning does not fully isolate because it ignores *state*. Workloads depend on the host kernel for functionality, creating a dependency on shared kernel objects. For instance, two containers that are assigned separate CPU cores and memory limits via cgroups may still interfere if they repeatedly access the global inode cache or filesystem journal, despite no violation on assigned resource limits. To capture this, we define *Data Isolation* as the property ensuring that the kernel objects accessible by one workload are structurally disjoint from those of another. We quantify this via the workload's *Data Footprint*: the set of all kernel objects it accesses during execution.

Understanding and measuring this shared state are also critical requirements for evolving the design of isolation mechanisms. Many emerging designs can be viewed as pursuing *state minimalism*, aiming to reduce, restrict, or privatize kernel state that is exposed across workloads. Specifically, Address Space Isolation (ASI) [58, 59] restricts the sensitive kernel state accessible to the currently executing process to prevent leakage, while other approaches aim to create private copies of highly shared data for each container instance to eliminate inter-tenant interference [84]. Similarly, Library OS designs [117, 128, 105, 134] migrate OS services out of the shared host kernel, thereby reducing shared kernel state. However, enforcing these boundaries within a monolithic kernel requires identifying existing structural dependencies. The kernel state cannot be effectively partitioned or privatized without understanding which objects are implicitly shared. A systematic study of kernel object sharing is therefore essential not only for assessing current isolation guarantees, but also for providing the empirical foundation needed to design stronger isolation mechanisms.

4.1.2 Performance vs. Efficiency

Stronger forms of isolation often provide increasingly more private resources and fewer shared resources. At the extreme, running a workload on separate physical machines

provides extreme isolation, while workloads running in standard OS processes share most kernel resources. Virtual machines lie in the middle and give workloads a private guest kernel managing partitioned memory and often dedicated CPUs. We observe that isolation is often opposed to efficiency: as a platform increases isolation, it decreases the amount of sharing across workloads, which reduces the opportunity for interference but comes at the cost of efficiency. This provides a strong motivation to find the *right level of isolation* for a workload — using a too-strong platform wastes resources, while a too-efficient platform with sharing increases the risk of interference. Table 4.1 shows the resources that are shared/isolated across different isolation platforms.

Moreover, the *right* level of isolation depends on the workload. Workloads that rely minimally on kernel services may safely leverage the high efficiency and ease of sharing (*e.g.,* shared libraries) of a less isolated platform. Workloads that frequently interact with kernel subsystems are more sensitive to shared kernel state. However, we lack visibility into a workload's *data footprint* on a platform to determine which platform is appropriate so it is difficult to assess if a platform's efficiency gains come at the cost of detrimental sharing.

4.1.3 Synchronization as an Identifier of Sharing

Interference and synchronization. Interference implies that some shared resource between workloads acts as an agent of interference. Our key insight is that operating systems *synchronize all access to shared resources*, whether a physical resource (memory, devices) or a logical resource (an object). For instance, concurrent processes must synchronize via memory allocator locks when requesting physical pages, just as they use inode locks when accessing a shared file to ensure consistency. Thus, any interference between workloads from shared objects will be controlled by synchronization. This synchronization ranges from private locks protecting local process state, to shared locks guarding global resources, like `tasklist_lock`, which serializes access to the global task list. Processes sharing memory

synchronize access to physical pages and allocator state; processes sharing a device (*e.g.*, a network or sound card) synchronize through device queues; and processes accessing files synchronize through file-system objects.

Moreover, not all kernel state is equally shared. Many subsystems rely on per-CPU data structures, which localize state to individual cores and avoid synchronization across CPUs. While these structures may be accessed by different threads over time as workloads are scheduled on the same CPU, this temporal sharing does not require locks and is designed to avoid cross-processor synchronization. By design, these structures avoid the type of cross-workload interference we measure. Interference only occurs when these local resources are exhausted, and the kernel must coordinate through globally shared pools (*e.g.*, refilling a per-CPU slub cache), at which point synchronization on shared kernel objects occurs and is captured by our analysis.

We note that OS design can lead to *incidentally* shared data, which means processes logically access different data, but the OS design requires access to shared data, similar to false sharing with locks. For example, workloads accessing different files on a shared file system, even if they access different files, may synchronize access to shared caches such as the Linux `dcache` or `inode` cache and access shared hash structures. Figure 4.1 shows examples (code snippets) of private, shared, and incidentally shared locks in the Linux kernel.

Locks and synchronization. Shared kernel resources require synchronization to ensure a safe and correct order of execution of concurrent processes in the system. However, from an isolation perspective, not all synchronization mechanisms contribute equally to interference. Read-only sharing does not lead to interference, as each process can logically operate on a copy of data without affecting other processes. Our goal is to quantify interference arising from shared kernel state, and thus we focus on synchronization mechanisms that are most likely to impact isolation.

The Linux kernel employs two categories of synchronization primitives: (a) *Lock-based*

synchronization, including spinlocks (and variants such as reader–writer locks and seqlocks), mutexes, and semaphores; and (b) *Lockless synchronization*, including atomic operations, Read-Copy Update (RCU), and memory barriers. A key difference between these mechanisms is that lockless synchronization is typically non-blocking [49, 60, 145]. For example, RCU [106] is optimized for read-heavy sharing, allowing readers to access data concurrently without blocking even during updates. Therefore, interactions in lockless paths are transient and usually allow execution to proceed in parallel, making them less likely to expose blocking-induced performance interference. However, from a data-isolation perspective, lockless mechanisms can still represent shared kernel state that is not captured by our approach. We do not expect this limitation to substantially affect subsystems that primarily rely on lock-based synchronization.

In contrast, locks enforce mutual exclusion to serialize access to shared kernel data; hence, we believe that they impact interference and isolation more. This distinction motivates our focus on lock-based synchronization as a proxy for shared kernel data access. Therefore, we use lock-based synchronization as our primary proxy for sharing. While this approach does not capture all forms of sharing, it provides a starting point for quantifying data isolation through kernel synchronization.

Limitation. We note that our approach does not currently capture lock-free synchronization mechanisms such as RCU and atomic operations. These are generally non-blocking and may have limited impact on performance isolation; RCU is optimized for read-heavy sharing, allowing readers to access data concurrently without blocking even during updates. However, from a data-isolation perspective, they can still introduce forms of data sharing not captured by LIMA. We do not expect this limitation to affect results for subsystems that primarily use lock-based synchronization.

	Lock name	Struct name	Struct path	Acq/sec
lxc	mapping->private_lock	address_space	include/linux/fs.h	527
	journal->j_list_lock	journal_s	include/linux/jbd2.h	444
	sbi->s_orphan_lock	ext4_sb_info	fs/ext4/ext4.h	333
	inode_hash_lock			174
	journal->j_state_lock	journal_s	include/linux/jbd2.h	78
	others			11.53
fc	journal->j_state_lock	journal_s	include/linux/jbd2.h	1214
	lruvec->lru_lock	lruvec	include/linux/mmzone.h	1.25
	zone->lock	zone	/include/linux/mmzone.h	0.3
	journal->j_state_lock	journal_s	include/linux/jbd2.h	0.12
	journal->j_wait_commit	journal_s	include/linux/jbd2.h	0.03
	others			0.11

Table 4.2: Top-5 locks and other locks for *lxc* and *fc* for *listdirs*.

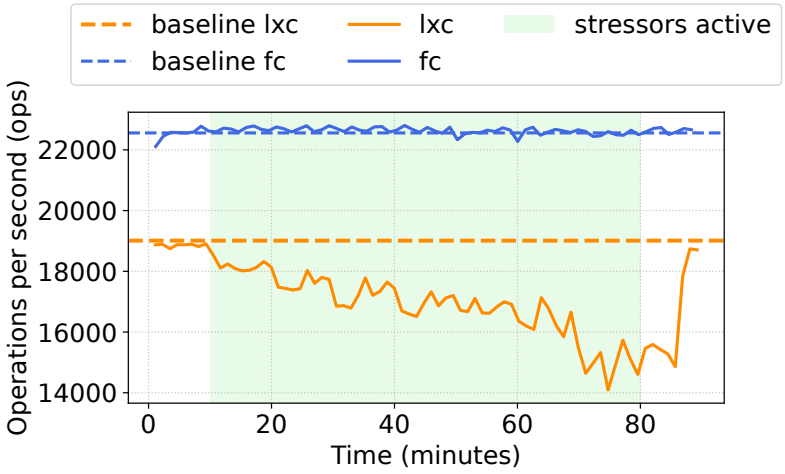


Figure 4.2: Impact of interfering workloads on *listdirs* performance over time.

4.2 Case Study

Isolation platforms fundamentally shape how workloads interact with host kernel state. In this section, we show that the same workload exhibits different kernel data footprints depending on the choice of platform architecture, revealing distinct interference vectors, *i.e.*, shared kernel objects through which concurrent workloads interact and interfere. We use *listdirs* from filebench [132], which repeatedly enumerates directory entries. It exercises the filesystem metadata path and stresses both fine-grained per-object locks and coarse global locks, making it well-suited for studying the *shared data footprint* across different levels of granularity. We use this workload to highlight several key aspects of object-level sharing: the impact of isolation platform designs, the role of incidental locks on interference, and

the impact of co-location with other workloads.

We study native Linux containers built on cgroups (*lxc*), as implemented by Docker's *runc* container engine [113, 14] and Firecracker (*fc*) [36] because they represent two distinct points in the isolation design space. *lxc* provides OS-level isolation by sharing a single host kernel across workloads, while *fc* uses a microVM architecture with a dedicated guest kernel, yet still relies on the host for key functionalities. This contrast lets us examine how platform structure shapes shared kernel state.

We run two concurrent instances of *listdirs* on the same platform and record the lock accesses for each instance. We then identify shared locks by intersecting the sets of unique lock addresses accessed by both workloads. Using lock addresses, we can distinguish between global locks and per-object locks. For example, `inode_hash_lock` is a global lock; `inode->i_lock` is a distinct lock instance within every open file. Our methodology counts locks as shared only when both workloads access the same underlying object instance, ensuring we capture sharing of specific kernel data.

While it is widely understood that microVMs offer stronger isolation, our goal is to move beyond qualitative wisdom and quantify the specific kernel structures that form the interference vector in each platform. In Table 4.2, we show the top shared locks (locks held in common across workloads) accessed by these platforms and their lock access rates (in locks acquired/sec).

The results reveal a structural difference in how these platforms expose the kernel. Under *lxc*, the workload interacts with a broad set of objects, spanning security (*e.g.,* `aa_buffers_lock`), process management, and filesystem objects. *fc* accesses a narrower set of memory-management objects (*e.g.,* `struct lruvec` holding candidates for page replacement via `lru_lock`), consistent with *fc*'s guest kernel performing most memory management. *fc* concentrates activity on a few shared objects at high frequency, whereas *lxc* spreads sharing across many objects, quantifying the narrower interface of microVMs compared to containers.

A few kernel objects dominate sharing. We observe a few highly accessed shared objects on both platforms that abstract managing shared hardware resources. For memory, we see access to the memory zones, `struct zone`, which organizes and tracks state for physical memory zones. For CPUs, we see the scheduler's `tasklist_lock` protecting the global task list, and for the file system we see accesses to `journal_s`, which manages the journal state, synchronized via `j_list_lock` and `j_state_lock`. These recurring shared objects highlight that kernel-level sharing centers on a few critical structures that dominate the shared data footprint.

Incidental sharing matters for interference. We define incidental sharing as cases where workloads logically access disjoint data but, due to OS design, synchronize on shared kernel structures. Locks related to incidental sharing like the global `inode_hash_lock` which protects the inode cache hash table show a significant access rate (174 times/sec for *lxc*). Because such sharing arises from kernel data structures like hash tables rather than application logic, it is difficult for tenants to anticipate or avoid. This implies that even workloads without obvious overlap in their data paths may still contend for incidental locks, leading to interference.

Object sharing influences workload performance. Figure 4.2 shows the impact of object-level sharing on performance. We co-locate *listdirs* with interfering workloads (called *stressors*), running the same *listdirs* workload to create pressure on overlapping objects. We start one *stressor* at 10 minutes, adding additional ones every 10 minutes for 70 minutes. While *fc* exhibits a high level of object sharing in our traces, we observe visible performance degradation only for *lxc*. The lock tracing data show far more sharing of file system data (the top 5 locks acquired in *lxc* are all related to file system), and all are acquired more than 100 times/sec. In contrast, with *fc* only two journal locks are accessed, making it much more robust to competing workloads.

This case study illustrates how a single workload can reveal the role of object-level sharing in software-level interference, reflecting platform design, which kernel paths are

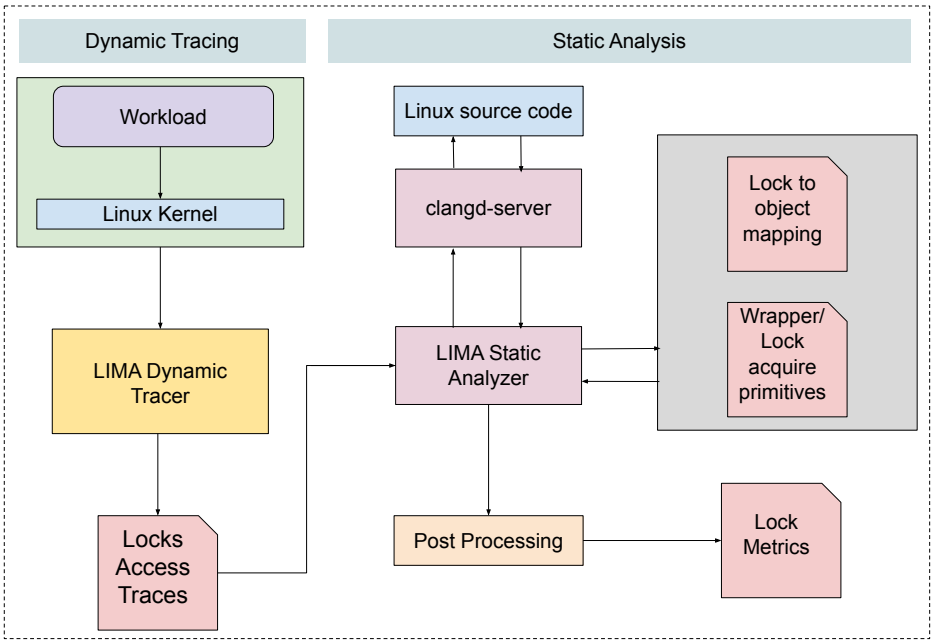


Figure 4.3: Overview of *LIMA* workflow. In the dynamic tracing phase, *LIMA* collects lock accesses across workloads, and the static analysis phase completes the lock-to-object mapping.

Lock Type	Lock Primitives
Spinlock	_raw_spin_lock, _raw_spin_lock_irqsave, _raw_spin_lock_irq, _raw_spin_lock_bh, _raw_spin_trylock, _raw_spin_trylock_bh,_raw_spin_lock_nested, _raw_spin_lock_irqsave_nested,
Read-write lock	_raw_read_lock, _raw_write_lock, _raw_read_lock_bh, _raw_write_lock_bh, _raw_read_lock_irq, _raw_write_lock_irq, _raw_read_lock_irqsave, _raw_write_lock_irqsave, _raw_read_trylock, _raw_write_trylock, _raw_write_lock_nested
Mutex	mutex_lock_nested/mutex_lock, rt_mutex_lock_nested/rt_mutex_lock, mutex_trylock, rt_mutex_trylock, mutex_lock_interruptible_nested
Semaphore	down_read, down_write, down_read_trylock, down_write_trylock, down_read_nested, down_write_nested, down_read_killable, down_write_killable, down_read_killable_nested, down_write_killable_nested, down_read_interruptible

Table 4.3: Lock primitives probed by *LIMA* dynamic tracer.

exercised, and incidental locks. These findings highlight that object-level sharing is common during workload execution on modern isolation platforms.

4.3 LIMA Design

We implemented a data-collection tool named *LIMA* that identifies the kernel objects accessed by workloads running on different isolation platforms on Linux, enabling object-level analysis of shared kernel state. To achieve this, *LIMA* is designed with the following

goals:

- *Workload-aware measurement*. Capture runtime lock activity as object sharing depends on real execution, rather than potential code paths.
- *Object-level attribution*. *LIMA* should reveal the underlying objects that those locks protect. This is important so that data sharing can be linked to specific kernel structures.
- *Cross-platform comparability*. Provide a uniform mechanism that works across diverse isolation platforms to allow systematic comparison of their isolation boundaries.
- *No modifications to the kernel*. Ability to run on the host without any modifications to the kernel code.

LIMA combines two complementary components: it traces lock acquisitions dynamically using eBPF and uses static analysis to resolve where in the code each acquisition occurs and, more importantly, what objects the lock protects. We show an overview of *LIMA* in Figure 4.3. These components enable attribution of runtime shared kernel state access to specific kernel objects.

Challenges. Building *LIMA* revealed several challenges that make object-level sharing analysis non-trivial. First, dynamic tracing via eBPF may produce incomplete stack traces, as inlining and tail-call optimizations [78, 80] can truncate frames before the actual lock acquisition. Second, system-wide tracing introduces noise, since locks may be acquired by unrelated processes or in interrupt contexts (*e.g.,* timers or softirqs), requiring careful filtering to isolate workload-induced sharing. Finally, tracing itself incurs overhead, which can perturb execution and potentially miss short-lived or low-frequency lock events.

Limitations. Despite our mitigations, *LIMA* has limitations. It does not trace lockless primitives such as RCU or atomics. Additionally, it cannot attribute lock activity in an interrupt context to a specific workload. *LIMA* can identify some forms of sub-structural locking, such as locks embedded in nested structures, but it cannot directly distinguish which fields are protected when multiple locks protect different fields within the same structure; in such cases, we rely on kernel documentation and source code when needed.

Hardware-level sharing is out of scope: while we reduce its effects through techniques such as CPU pinning and Cache Allocation Technology (CAT), residual contention in caches, memory, or I/O remains outside the scope of this work. Finally, *LIMA* currently does not resolve all cases where locks are acquired through macros or where object variables are reassigned to local variables within function bodies.

Generality. While kernel changes (*e.g.*, optimizations for multi-core scalability or restructuring of shared objects) may affect the degree and distribution of sharing, *LIMA* is designed to be reusable across kernel versions.

4.3.1 LIMA Dynamic Tracer

We extended and modified the eBPF-based tool *klockstat* [118] to implement the dynamic tracing component of *LIMA* to trace dynamic lock usage. *LIMA* traces lock acquire and release functions of mutexes, semaphores, spinlocks, and reader-writer spinlocks. By instrumenting these primitives, *LIMA* provides comprehensive coverage of the kernel's lock-based synchronization mechanisms. Table 4.3 lists all the lock primitives we probe with LIMA.

LIMA dynamic tracer produces a trace of every lock acquired with lock name (the kernel symbol associated with the lock variable), kernel stack trace, acquire time, hold time, and count of occurrences of that stack trace with that lock. In addition, we instrument lock initialization routines to learn when a lock address is reallocated for a new lock. We process this trace to calculate metrics such as the number of *shared locks* — locks acquired in common across workloads — accessed for different workloads and *private locks* — unique and non-shared locks acquired by each workload.

Dynamic analysis produces a trace of all the locks acquired by a running workload. Below is an example of the trace output. The stack details are stored in a different file and matched with the `stack_id` to get the stack traces with an acquisition.

Functionality	Description
Symbol Generation	Extracts symbol details (name, type, signature, line range) to map locks to structs and list global locks (≈2488).
Outgoing Calls	Uses function info to identify outgoing calls and locate lock acquisition in incomplete stacks.
Incoming Calls	Identifies lock wrapper functions (335 found) from incoming call info, used as stop points in stack resolution.
AST	Builds abstract syntax trees to disambiguate "generic" locks (named "lock") and link them to the right object.
Get Definition	Returns file path and line number of a lock, matched with struct ranges from Symbol Generation.

Table 4.4: Core functionalities of *LIMA*'s static analysis.

PID	addr	name	count	process	stack_id
38296	0xffff888	&pipe->mutex	1	mmap04	324596

4.3.2 LIMA Static Analyzer

LIMA static analyzer processes the dynamic lock traces and completes the mapping of lock names to kernel objects. We implemented the *LIMA* static analyzer using the clangd [54] language server [34]. clangd is widely used in many IDEs for different functionalities (*e.g.*, symbol indexing and outgoing call analysis), and it provides convenient APIs for querying code properties.

As summarized in Table 4.4, *LIMA* static analyzer combines symbol generation to index lock definitions, call graph analysis (incoming/outgoing calls) to resolve incomplete dynamic stack traces and wrapper functions, and AST traversal to disambiguate generic lock members. These core functionalities enable the precise resolution of lock-to-object mappings, *i.e.*, identifying the parent kernel structure (*e.g.*, `struct inode`) that contains a given lock member (*e.g.*, `i_lock`). This transforms lock names into semantically meaningful object references, as described in the steps below.

Input. *LIMA* static analyzer takes three primary inputs: the Linux kernel source code, the dynamic lock access traces collected at runtime and the kernel binary (for address resolution). It queries the indexed source code via the clangd server to access the core analysis functionalities.

Object Mapping. The core functionality of *LIMA* is to map an instance of lock acquisition to the object it protects. To do this, we rely on the call stack captured during tracing to identify the code location responsible for acquiring the lock. The stack allows us to determine

which function invoked the locking primitive and, consequently, which object instance is being accessed. This involves:

Incomplete stack trace resolution: As noted in challenges, dynamic tracing often truncates stack traces, hiding the actual lock acquisition function. To recover this missing context, we leverage a database of *Outgoing Calls* (outgoing_calls_db) which we have built across the kernel source. Given an incomplete dynamic trace, we take as input the lock name, file, and function where tracing stopped, and then initiate a breadth-first search from that last function over outgoing_calls_db. The search terminates when we reach either a standard lock primitive (Table 4.3) or a known lock wrapper, a helper function (*e.g.,* mmap_read_lock) that encapsulates the locking logic, identified via our *Incoming Calls*–based wrapper database.

Object resolution: We generated a database for global locks (global_locks_db with 2488 entries) using *Symbol Generation* that contains information about the lock, as in this example:

lock_name	**lock_type**	**file**
tasklist_lock	rwlock_t	include/linux/sched/task.h

We also use *Symbol Generation* to create a mapping of locks to the `struct` within which they are defined (lock_struct_map_db), with details:

file	**struct_name**	**lock_type**	**lock_name**	**line_range**
mm/slab.h	kmem_cache_node	spinlock_t	list_lock	751-779

We start the search for lock-to-object mapping by querying these two files first. A trace may identify an acquisition of a variable named `lock`. However, because hundreds of different structures in the Linux kernel use the name `lock`, simple text matching is insufficient. For example, it does not distinguish whether a specific lock named `lock` belongs to `fs_struct, zone`, or any other object.

To resolve this, we use the AST file of the function where *generic* locks are acquired to get the line number associated with the lock variable, which we use to query *Get Definition* for

mode	workload	workload events	other events	interrupts
cold start	microbench	131 (2%)	5003 (87%)	714 (10%)
	funcbench	152 (2%)	5352 (84%)	1222 (14%)
	cloud	241 (3%)	6207 (80%)	1434 (17%)
steady state	microbench	107 (3%)	3502 (78%)	914 (19%)
	funcbench	97 (2%)	3271 (76%)	1443 (24%)
	cloud	147 (2%)	4732 (74%)	1530 (24%)

Table 4.5: Average traces collected across workload events, interrupts, and other events across all workloads and platform combinations.

the lock symbol to find the location of its definition. We use this to locate the corresponding object name by matching the line number to line_range (the lock line number should fall within this range for a match) and file in lock_struct_map_db, hence completing the mapping.

Output. At the end of the processing, *LIMA* generates a file containing end-to-end lock to object mapping containing:

file	object_name	lock_type	lock_name
fs/ext4/ext4.h	ext4_inode_info	rw_semaphore	i_data_sem

For all unique locks—identified by their primitive, lock name, function (which we use to match dynamic trace entries with the static lock-to-object mapping), and source file-acquired across all workloads analyzed (984 in total), we can map approximately 85% of them to their associated objects. Even when a lock cannot be fully resolved, our approach provides valuable contextual information about the subsystem and object involved by pointing to the line of lock acquisition in the kernel codebase.

4.4 Object Analysis Methodology

Our primary goal is to measure data interference arising from shared access to kernel resources. For each workload under test, we collect host kernel lock traces to understand how different isolation platforms use various kernel data structures. These traces allow

Microbench	Description
fork (stress-ng)	repeatedly creates and reaps processes.
futex (stress-ng)	stresses futex wait/wake with threads.
sysbench	finds prime numbers for a fixed duration.
mem (1MB)	allocates 16GB using 1MB allocation size.
mem (8KB)	allocates 16GB using 8KB allocation size.
createfiles (filebench)	measures file creation performance.
deletefiles (filebench)	measures file deletion performance.
listdirs (filebench)	lists the contents of a large directory.

Table 4.6: Microbenchmarks with their descriptions.

Cloud Workloads
data analytics, data caching, data serving, graph analytics, in-memory analytics, media streaming
FunctionBench
image processing, video processing, model training, cnn, rnn, face detection
stress-ng
fork, vm (mem), io, fstat, getdent

Table 4.7: Cloud, serverless workloads and stress-ng stressors analyzed by *LIMA*.

us to observe which kernel objects are accessed, how frequently they are accessed, and which objects are shared across co-running workloads. Importantly, our study focuses on logical data isolation rather than hardware performance interference; hence, we design our experiments to minimize hardware-level contention (*e.g.,* not exceeding local caches or disk bandwidth).

4.4.1 Experimental platform

We run experiments on CloudLab [12] c220g5 nodes (2.20 GHz Intel Xeon Silver 4114, 192GB Memory, Intel DC S3500 480 GB SSD, 1 TB 7200 RPM HDD, and 10Gbps NIC) running Ubuntu 22.04 with Linux kernel v6.1. We enable lock-related debugging configurations (*e.g.,* CONFIG_DEBUG_SPINLOCK) for collecting traces. We try to minimize interference through shared hardware: we disable SMT and isolate the LLC by enabling Intel's Cache Allocation Technology (CAT). We have 11-way LLC partitioning, and each Class of Service (COS) is assigned to one cache line. All cores on socket 0 have a one-to-one mapping with a COS.

4.4.2 Isolation Platforms

We run workloads on three platforms: (a) Native Linux container (*lxc*), (b) gVisor release-20250421.0 (*gv*) (we use the KVM platform; recommended for bare-metal [82]), and (c) Firecracker v1.10.1 (*fc*).

These platforms represent three distinct points in the isolation design space: *lxc* provides OS-level isolation using Linux namespaces and cgroups while directly sharing the host kernel. *gv* implements a user-space kernel (sandbox) that mediates most system calls and kernel interactions, reducing direct exposure to host kernel structures. *fc* executes each workload inside a lightweight virtual machine with its own guest kernel, thereby reducing host-kernel sharing at the cost of additional virtualization layers. All platforms use standard Linux mechanisms for CPU and memory resource control where applicable (*e.g.,* cgroups), while fc relies on the VM boundary to isolate kernel state in addition to host-level resource controls. By selecting these three platforms, we capture a spectrum from direct host-kernel sharing (*lxc*), to mediated kernel interaction (*gv*), to guest-kernel virtualization (*fc*), enabling comparison of how platform design shapes kernel object sharing.

4.4.3 Workloads

Our goal is to understand how the *data footprint* of workloads varies across a diverse spectrum of system behavior (*e.g.,* stressing particular subsystems, light/medium/heavy kernel subsystem usage). To capture these diverse cases, we chose workloads to measure different dimensions of kernel sharing.

Microbenchmarks. We choose a set of microbenchmarks from stress-ng [93] and Filebench [132]. We wrote our own memory microbenchmark that allocates/deallocates and touches 16GB of memory using mmap/munmap in a loop using different allocation sizes. These microbenchmarks stress different kernel subsystems to understand data footprints under targeted stress environments.

Serverless Workloads. To gain insights into short-lived object usage patterns in a FaaS environment, we analyze lightweight serverless functions from FunctionBench [91], selecting applications (image/video processing), ML training (logistic regression), and ML serving (face detection, CNN, RNN).

Cloud Workloads. To capture the data footprints of long-term and heavy workloads, we run CloudSuite cloud workloads [13] as shown in Table 4.7. We run all workloads in standalone mode [89], *i.e.*, all the services and components (like client and server) of the workload run in a single instance of the platform.

For microbenchmarks, we run two instances of the same workload simultaneously on the same isolation platform. For each of six FunctionBench and six CloudSuite workloads, we pair the workload with one of five different stress-ng *stressors* listed in Table 4.7. This pressures different kernel subsystems for a total of 60 combinations. Filebench requires disabling ASLR, which is not supported on *gv*, so we could not evaluate it on *gv*.

For microbenchmark and serverless workload tracing, each workload is executed for a fixed duration. For cloud workloads, we terminate workloads after 30 minutes. We execute the workload separately for each lock type, tracing one lock type at a time to obtain cleaner traces and improved lock coverage. We collect traces over three iterations per lock type to maximize coverage of unique locks. We do not trace platform startup phases.

To distinguish interference sensitivity during the startup phase, which matters for bursty or short-lived workloads, from steady-state sharing that dominates long-running/persistent services, we trace in two modes: (1) *cold start* and (2) *steady state*. In *cold start* mode, tracing begins at workload launch to capture initialization effects (cache/page faults, inode/dcache population). In *steady state* mode, tracing begins after the workload has run for some time, capturing the steady-state interference pattern. For cloud workloads in *steady state* mode, we completely execute the workload once and trace in the second iteration.

4.4.4 Analysis Outputs

We use lock traces to determine the following:

- *Shared locks*: We identify shared locks by their lock addresses and report the median count across runs. We calculate the median per lock type (spin, mutex, semaphores, *etc.*) and sum across lock types.
- *Average lock acquire rate*: We report average lock acquire rates, $\text{Rate}_\ell = \frac{\sum_{r=1}^{N} \text{lock count}_\ell^r}{\sum_{r=1}^{N} \text{workload execution time}^r}$, where lock count is the number of times a lock ℓ (uniquely identified by lock name and lock type) was acquired, r is the iteration number and N is the total iterations. Tracing slows execution, so the rate is lower than for an uninstrumented kernel.

Together, these metrics capture the breadth and intensity of the shared data footprint for a workload. A workload may exhibit *wide and shallow* sharing – touching many kernel objects at low rates – or *narrow and deep sharing* – concentrating activity on a small number of hot objects. These patterns reveal which specific kernel objects become the structural points through which isolation breaks on each platform. Table 4.5 shows the average traces collected. We use the workload event traces for our analysis.

In our analysis, we use lock acquisitions as a proxy for accesses to shared kernel objects. Since kernel locks are tightly associated with specific data structures, we use the terms lock and object interchangeably when discussing sharing, with the understanding that a lock represents the kernel object whose access it serializes.

4.5 Object Level Interference Analysis

Our evaluation goals are:

- How do object-level sharing and hence, *data isolation* vary across isolation platforms?
- What kernel objects constitute the *shared data footprint* of a workload, and how does this footprint vary across different classes of workloads?
- Which kernel subsystems dominate object-level sharing across platforms?

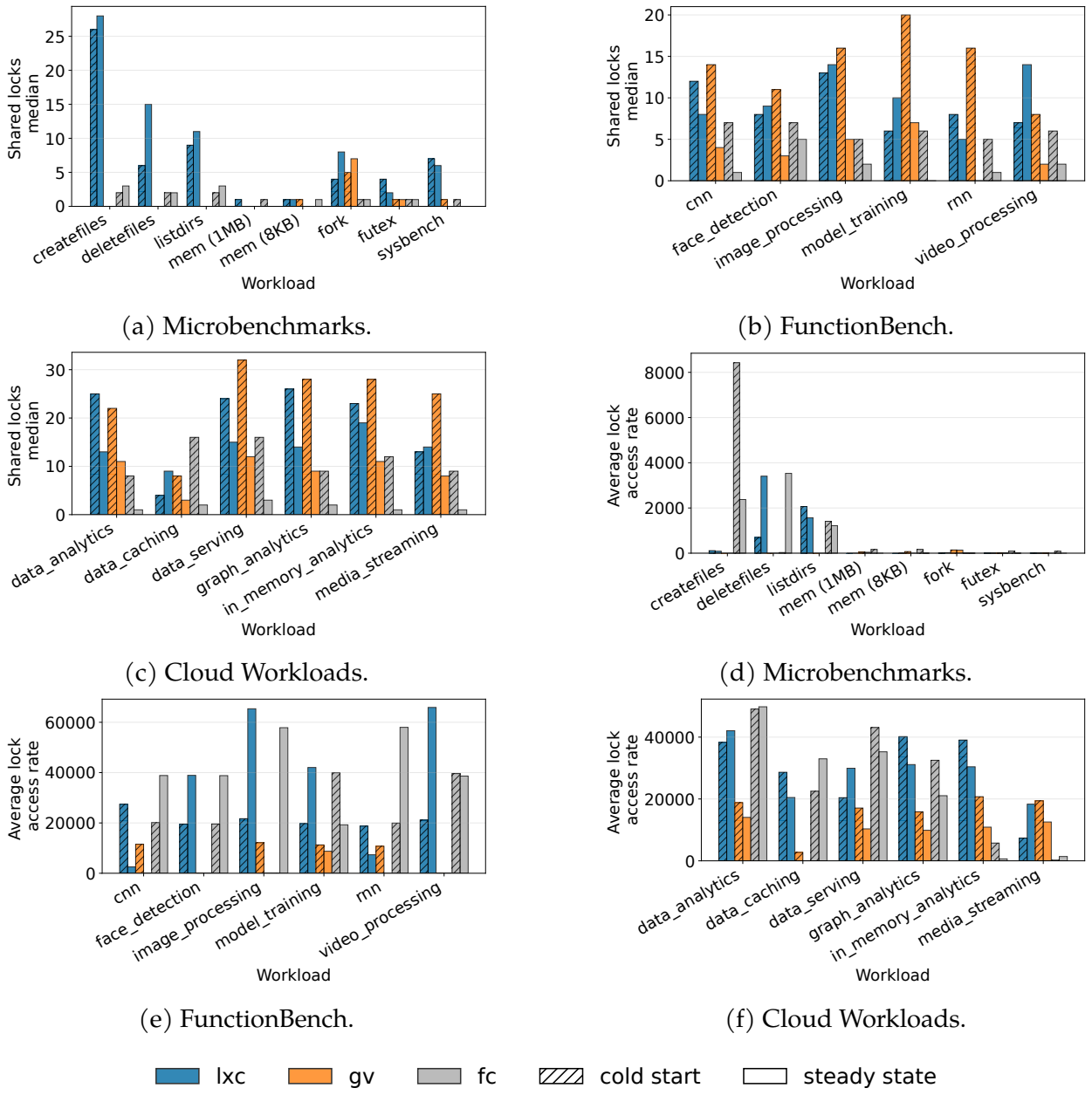
lock name	description	Acquires/sec (trace points)	struct name	struct path	static global
&journal->j_state_lock	Ext4 journaling: serializes updates to the journal state and transaction lifecycle.	57982 (279)	journal_s	include/linux/jbd2.h	
&root->kernfs_rwsem	kernfs/sysfs/cgroupfs root rwsem: protects the global directory tree for kernfs operations.	18684 (8)	kernfs_root	fs/kernfs/kernfs-internal.h	
proc_subdir_lock	synchronize access to the directory structure of the /proc virtual filesystem	12067 (18)			Y
&journal->j_list_lock	Ext4 journaling: protects the lists of running/committing transactions.	10302 (156)	journal_s	include/linux/jbd2.h	
&journal->j_wait_updates	Ext4 journaling: synchronizes waiters for journal updates and transaction progress.	10208 (50)	journal_s	include/linux/jbd2.h	
aa_buffers_lock	AppArmor security module to protect a global pool of memory buffers	5953 (35)			Y
&rq->__lock	Scheduler runqueue spinlock: serializes enqueue/dequeue and scheduling decisions per CPU.	2814 (33)	rq	kernel/sched/sched.h	
&ei->i_raw_lock	ext4_inode_info: synchronizes inode-specific ext4 metadata updates.	2043 (7)	ext4_inode_info	fs/ext4/ext4.h	
tasklist_lock	Process management: global rwlock protecting task list enumeration and updates.	839 (91)			Y
&mapping->private_lock	VFS/address_space: protects private/mapping-specific data structures.	742 (165)	address_space	include/linux/fs.h	
&zone->lock	Memory allocator: per-zone lock protecting free area lists and page allocator metadata.	381 (301)	zone	/include/linux/mmzone.h	
&sbi->s_orphan_lock	Ext4 superblock: protects the list of orphaned inodes (unlink/truncate crash safety).	333 (39)	ext4_sb_info	fs/ext4/ext4.h	
&lruvec->lru_lock	Memory reclaim: protects per-memcg/node LRU lists during page reclaim/rotation.	273 (148)	lruvec	include/linux/mmzone.h	
&sbi->s_es_lock	Ext4 extent status tree: protects in-memory extent status caches for files.	250 (38)	ext4_sb_info	fs/ext4/ext4.h	
per_cpu_ptr (&cgroup_rstat_cpu_lock, cpu)	cgroup stats (per-CPU): per-CPU lock for resource accounting updates.	235 (43)			
&sbi->s_md_lock	Ext4 superblock: serializes modifications to superblock-wide metadata fields.	230 (33)	ext4_sb_info	fs/ext4/ext4.h	
inode_hash_lock	VFS inode hash: global lock protecting the inode hash table (lookup/insert/remove).	174 (61)			Y
&s->s_inode_list_lock	Superblock inode list: protects inode lists associated with a superblock.	166 (46)	super_block	include/linux/fs.h	
&bgl->locks[i].lock	Ext4 block group lock: serializes access to per-group allocation and metadata.	131 (54)	ext4_sb_info	fs/ext4/ext4.h	
shmem_swaplist_mutex	protects global list of shmem inodes that have swap space allocated	129 (54)			Y
tasklist_lock	Process management: global rwlock protecting task list enumeration and updates.	59 (66)			Y

Table 4.8: Top shared locks across our collected traces.

Answering these questions links object-level behavior to system-level isolation, revealing how platforms shift interference and identifying sensitive workloads and critical kernel subsystems.

4.5.1 Platform Impact on Object Sharing

We first quantify the extent of kernel-level sharing across workloads and platforms (§4.5.1.1), then attribute these events to specific kernel objects driving cross-workload interference (§4.5.1.2).



(a) Microbenchmarks.

(b) FunctionBench.

(c) Cloud Workloads.

(d) Microbenchmarks.

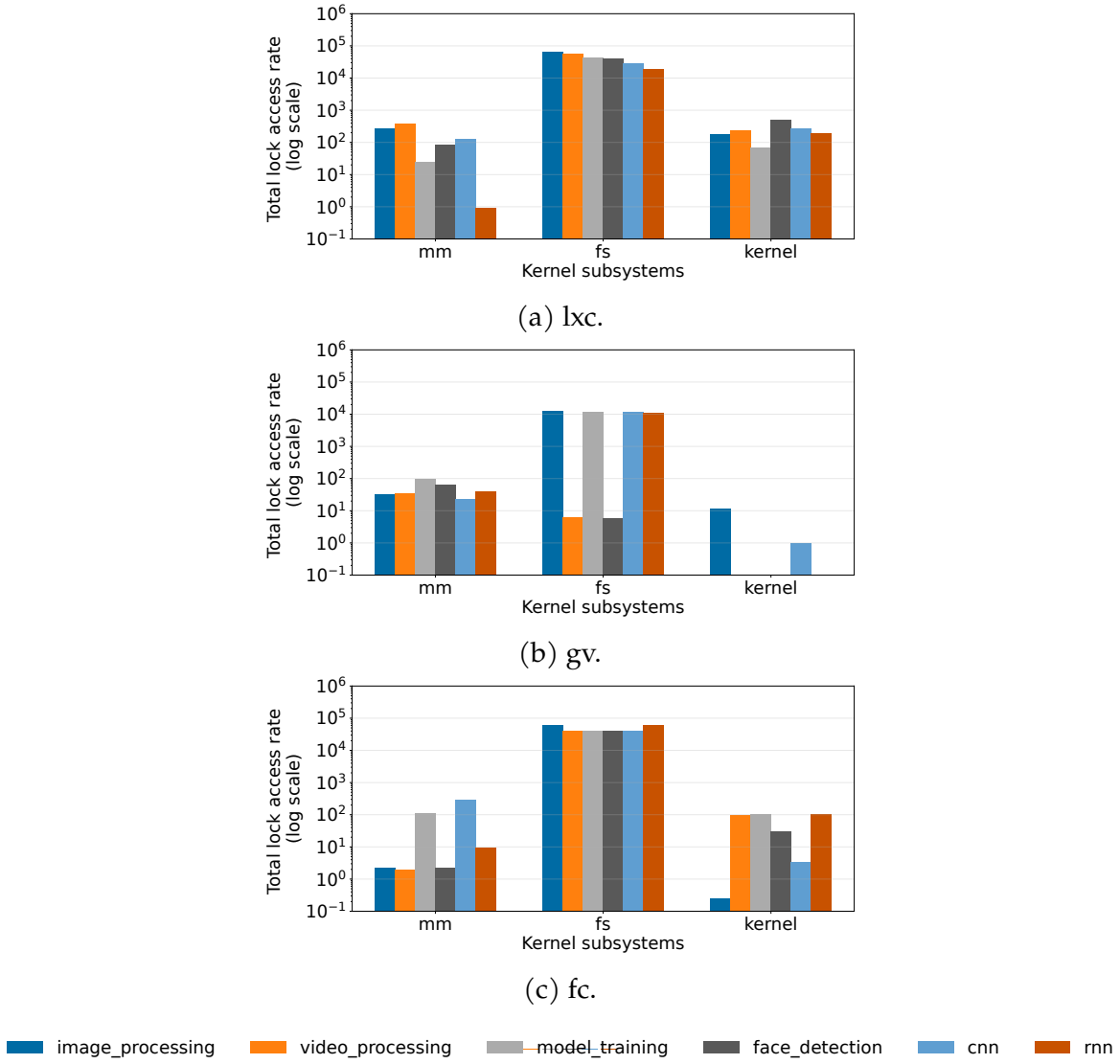
(e) FunctionBench.

(f) Cloud Workloads.

Figure 4.4: Figures 4.4a–4.4c show the median aggregate lock counts and figures 4.4d-4.4f show average access rates across platforms for each workload category.

4.5.1.1 Sharing magnitude

We use two metrics to characterize the sharing magnitude of different workloads on different platforms: (a) the median lock count, calculated by summing the median access count for each lock type (spinlock, mutexes, *etc.*) across runs, and (b) the average lock access rate



(a) lxc.

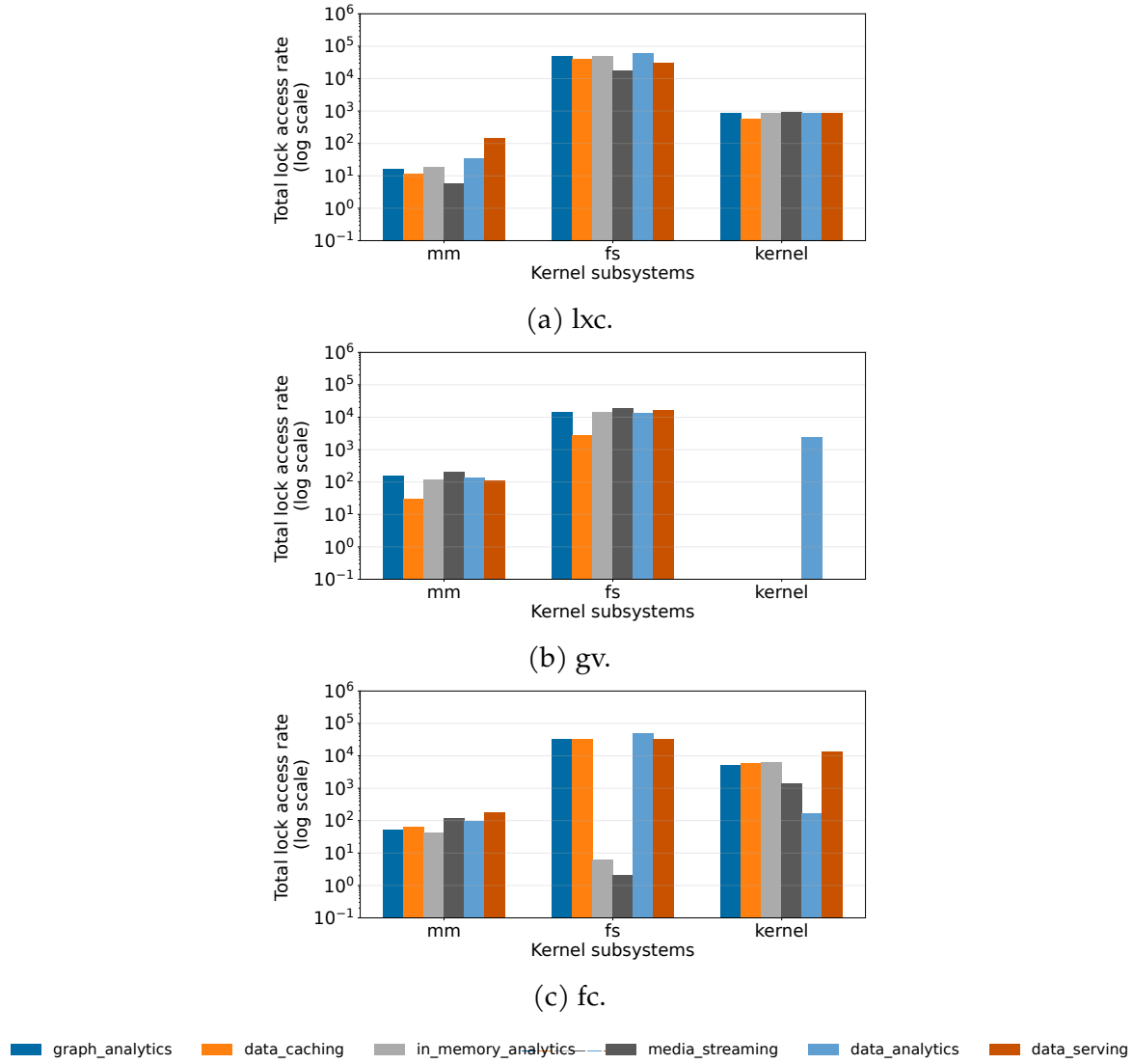
(b) gv.

(c) fc.

Figure 4.5: FunctionBench total lock access rate by subsystems. Most workloads show high usage in *mm* and *fs* across platforms.

across all shared locks. These metrics (Figure 4.4) are reported per platform and mode (cold start and steady state), with FunctionBench and cloud workloads aggregated over all stress-ng *stressors*. While this subsection quantifies sharing, we provide a detailed analysis of the specific kernel objects in §4.5.1.2.

Overall, *lxc* exhibits the broadest sharing footprint, acquiring 129 locks for microbenchmarks, 114 for FunctionBench, and 199 for cloud workloads, roughly 2.5× to 6× more unique locks than *fc*. In contrast, *fc* has the narrowest object exposure, with 21, 47, and 80



(a) lxc.

(b) gv.

(c) fc.

Figure 4.6: Cloud workloads total lock access rate by subsystems. Most workloads show high usage in *mm* and *fs* across platforms.

locks in the same categories, respectively. *gv* closely follows *lxc* for FunctionBench (106) and cloud workloads (197). Excluding filebench for a fair comparison of microbenchmarks (as filebench did not run on *gv*), *lxc* acquires 35 shared locks, *gv* (16), and *fc* (7), confirming that *lxc* maintains the broadest footprint while *fc* remains the narrowest. These differences reflect platform design and translate into distinct interference vectors: a broader sharing footprint increases the number of kernel objects through which co-located workloads can interact. This widespread sharing increases the likelihood of cross-workload influence and

interference, making isolation guarantees weaker and harder to reason about.

Looking at the total average lock access rate for sharing intensity, a different pattern emerges. For microbenchmarks, *fc* shows the highest average lock access rate at 17.5K acquires/sec, more than double that of *lxc* at 8K. However, Filebench did not run on *gv*. To enable a fair cross-platform comparison, we recompute the average rate excluding filebench workloads where *fc* has a rate of 509 acquires/sec, *gv* at 445, and *lxc* at 29. We suspect that LXC uses code paths with fewer shared objects, while the virtualization layer of *gv* and *fc* causes repeated accesses to shared locks. Overall, we see a narrow but deep sharing pattern for *fc*. For FunctionBench, *fc* again leads with the highest access rate at 391K, slightly exceeding *lxc* at 350K, with *gv* far behind at 55K. For cloud workloads, *lxc* has the highest rate at 346K, followed by *fc* at 294K, while *gv* trails substantially at 152K.

This high sharing intensity of *fc* stems from its reliance on the host for core functionality. Although workloads execute inside a guest kernel, *fc* relies on (i) a KVM-based hypervisor layer and (ii) virtio-backed devices for block and network I/O, both of which ultimately depend on host kernel subsystems for memory management, scheduling, and filesystem operations. As a result, memory provisioning (via host page allocation and reclamation), virtual CPU scheduling (via host runqueues), and host-backed filesystem journaling (as virtio-blk forwards guest I/O to the host kernel) remain shared objects. While this design reduces the number of exposed host objects relative to *lxc*, it does not eliminate sharing; instead, *fc* narrows object breadth while concentrating access on a smaller set of shared host objects.

These results show that platforms differ both in how many kernel objects they share and also in how intensely they access them. While *fc* uses separate guest OSes, it still shows high sharing due to the added virtualization layers, introducing shared kernel objects that become sharing hotspots. In contrast, *gv* intercepts and handles system calls in userspace, aggregating resource acquisition in its user-mode kernel and invoking the host only for essential operations. As a result, *gv* achieves both narrower and lower-intensity sharing by

shifting kernel functionality into its user-mode kernel, verifying its design goal of reduced kernel reliance.

Moreover, we observe a higher lock count and access rate across most workload–platform combinations in *cold start* mode (with some exceptions), reflecting initialization overhead. In comparison, *steady state* mode shows lower lock count and rate for some workloads, although some workloads maintain high rates, indicating persistent sharing of kernel objects (*e.g.*, file-metadata operations). Both *lxc* and *fc* exhibit high rates, mostly in *cold start* mode, with *fc* at times higher than *lxc*. This behavior highlights how platform design influences initialization behavior: additional virtualization layers introduce extra kernel interactions for setting up virtual devices, memory mappings, and filesystem metadata, increasing synchronization pressure.

Summary: Isolation platforms differ in both the *breadth* of shared kernel objects and the *intensity* with which those objects are accessed. *lxc* shows the widest sharing footprint, making its interference surface broad. *fc* reduces the number of exposed objects but concentrates activity on a smaller set of objects, producing a narrow and deep footprint. *gv* has the lowest access rates overall and comparatively lower lock counts for some workloads. *Cold start* amplifies these effects, highlighting that isolation concerns extend beyond performance latency to kernel-level sharing. Overall, these results show that isolation is not a binary property of a platform: platform design reshapes object-level sharing in different ways, exposing trade-offs that are hidden by coarse abstractions such as VM vs. container.

4.5.1.2 Highly shared objects and type of sharing

In practice, most objects are accessed only infrequently, but a few recur across workloads and runtimes, forming persistent sharing points. Table 4.8 shows the number of *trace points*, the number of distinct workload–stressor–platform–mode combinations (out of 2,785 total executions; microbenchmarks exclude stressors) in which a given lock (uniquely identified by its name, type pair (*e.g.*, `tasklist_lock`, read_rw)) is observed at least once.

Higher trace points indicate broader sharing and greater significance. The most significant recurring sharing falls into three categories.

1. Resource-driven serialization. These data structures manage physical resources that require strict serialization. In *memory management*, `struct zone` (via `zone->lock`, a dynamic per-object lock, which protects per-zone free area lists in the memory allocator) is extremely frequent (with 301 trace points). `struct lruvec` protected by `lruvec->lru_lock` (a dynamic per-object lock) appears in 148 trace points. The `lruvec` structure organizes pages into LRU lists that are central to global page replacement, page reclamation, and memory cgroup accounting. While `lruvec` structures can be partitioned by memory cgroups, page reclamation and page cache management require global coordination across workloads to maintain a consistent view of memory pressure. As a result, even workloads with small memory footprints contend for the same `lruvec` objects during reclamation or file I/O, making them a persistent source of cross-workload interference.

2. Filesystem and I/O serialization. These objects manage the filesystem and the integrity of persistent state. `struct dentry`, protected by `dentry->d_lock` (a fine-grained lock with 83 trace points), caches directory entries for accelerated pathname resolution. Because it maintains a shared view of filesystem namespace state, operations on different files may still converge on shared dentry structures and hash tables, making it a frequent synchronization point across workloads.

Similarly, journaling structures such as `struct journal_s` via `journal->j_state_lock` (279 trace points) and `journal->j_list_lock` (156 trace points) are prominent with high lock rates across workloads and platforms. The journal ensures crash consistency and ordering of the filesystem by coordinating metadata modification before disk commit. Since the filesystem maintains a single, consistent state for the underlying physical block device, these locks serialize metadata changes across workloads, making them dominant synchronization points.

Notably, we observe substantial filesystem synchronization under *fc* as well. Although *fc*

executes workloads inside a microVM with a separate guest kernel, storage I/O is exposed through virtio-blk devices backed by files on the host filesystem. While the guest has its own virtual filesystem, its persistent metadata updates are eventually serialized by the host's journaling structures. This design reduces direct kernel sharing but does not eliminate shared metadata paths, explaining why filesystem-related locks remain highly active even under VM-based isolation.

3. Global serialization. These objects maintain the global visibility of system state, creating sharing between logically disjoint workloads. `inode_hash_lock` synchronizes the *global inode hash* table to efficiently manage and access inodes, using a coarser global lock in the VFS layer. This represents incidental sharing: workloads accessing entirely different filesystems or isolated mount namespaces can still collide because the kernel maintains a single global hash table to index all inodes. Similarly, `tasklist_lock` (157 trace points), a static global `rwlock` in *process management*, serializes access to the *global system task list*, a coarse sharing point.

Other observations. We summarize the top-accessed objects (by acquires/sec) in Table 4.8. Cold start largely accesses the same set of highly shared objects as steady state, but with different acquisition rates. We also see that fine-grained and coarse-grained locks both contribute to sharing, though in different ways. Fine-grained locks such as `bgl->locks[i].lock` (per-block group) associated with `ext4_sb_info` protect specific filesystem metadata regions; although individually less intense, they create partitioned sharing surfaces tied to particular resource subsets. In contrast, global objects such as the global system task list (`tasklist_lock`) or the global page directory (`pgd_lock`) serialize a broad portion of kernel activity. These patterns show that interference stems both from frequent contention on fine-grained objects and from the breadth of coarse global objects—and across platforms, both persist.

Summary: A recurring set of shared objects, including both fine-grained per-object locks and coarse global locks, dominates sharing across workload-platform combinations. These

objects persist across isolation boundaries because they protect shared host functionalities such as memory allocation, filesystem consistency, and global kernel state. Workloads during *cold start* amplify kernel-level sharing across platforms, while in *steady state* many reduce but do not eliminate such sharing. This concentration suggests that mitigation should target recurring hot objects rather than broad system-wide redesigns. While some pressure can be reduced at the workload level (*e.g.*, disabling journaling for short-lived serverless tasks), stronger isolation requires privatizing or partitioning frequently contended kernel structures [84]. In particular, further partitioning allocator and journaling data structures could meaningfully reduce cross-workload interference.

4.5.2 Workloads' Sensitivity to Object Sharing Across Platforms

We now analyze how different classes of workloads impact the isolation of a platform design. We define *sensitivity* as the extent to which a workload's sharing profile varies across platforms in terms of breadth (how many shared objects), intensity (how frequently they are accessed), and composition (which kernel subsystems and data structures are involved).

4.5.2.1 Microbenchmarks

Across microbenchmarks as shown in Figures 4.4a and 4.4d, we observe that most workloads acquire relatively few locks in the *steady state*, with sharing dominated by initialization effects. Memory microbenchmarks are lean on locking: they show almost no acquisitions in *steady state* but spikes during *cold start*. After initialization, they operate largely within an established working set, and therefore show minimal sustained synchronization. During cold start, however, they converge on global memory management structures such as `zone` (via `zone->lock`, which protects per-zone free area lists in the page allocator) and `lruvec` (via `lruvec->lru_lock`, which organizes pages into lists for reclamation and memory pressure accounting). On *lxc*, allocation paths interact directly with the host, exposing the page

allocator and reclamation state throughout execution. While on *fc*, steady-state allocation is mostly confined to the guest, but cold start amplifies sharing due to VM setup and host-backed memory provisioning. *gv* lies in between, mediating allocation in userspace while still relying on host memory-management paths for faults and shared regions.

File-metadata operations (createfiles, deletefiles, listdirs) are the most lock-intensive across platforms. This behavior is dominated by filesystem structures such as `journal_s` (via `journal->j_state_lock` and `journal->j_list_lock`), `ext4_sb_info` (storing filesystem-specific state), `super_block`, *etc.* We also observe high acquisition rates even on *fc* for `journal_s` during *steady state*. Although *fc* runs workloads inside a microVM, storage I/O is ultimately handled by the host through virtio-based devices backed by host filesystem. As a result, filesystem metadata updates from the guest are translated into host-side operations that still interact with journaling state. This shows that filesystem hierarchy is difficult to isolate, even in microVMs; metadata operations remain a universal contention point regardless of the isolation boundary.

Process creation (*fork*) lies between memory and filesystem workloads in sharing sensitivity: it acquires more locks than memory workloads but at lower rates than file-metadata workloads, indicating coarse yet infrequent sharing. The objects involved differ by platform. Under *lxc*, *fork* interacts directly with host memory and scheduler structures, including page-cache objects such as `address_space` (represents the page-cache mapping of a file and tracks cached pages associated with an inode) and `list_lru_node`, as well as global task state via `tasklist_lock`, reflecting tight coupling to host memory-management and VFS subsystems. Under *gv*, *fork* primarily surfaces allocator-related objects such as `zone` and `swap_cgroup_ctrl` (tracks memory usage for swap control groups), since memory provisioning and scheduling still rely on host resources despite sandbox mediation. In contrast, under *fc*, most process management occurs within the guest kernel, avoiding host task-table updates, but *fork* still manifests through memory provisioning (`lruvec`) and scheduler structures such as runqueues (which serialize access to the CPU's per-core

runqueue via `rq->__lock`), especially during cold start. Thus, process creation introduces short-lived but unavoidable shared-state interactions across all platforms, making bursts of task creation structurally sensitive phases for isolation.

Finally, synchronization (futex) and compute (sysbench) benchmarks show the narrowest sharing footprint, though platform designs shift the point of contention. *fc* concentrates sharing on scheduler structures like `rq->__lock` to manage virtual CPUs, while *lxc* exposes a broader range of host VFS and memory objects. In contrast, *gv* limits interaction primarily to host-side memory allocators via the `zone->lock`, as its sandbox layer shields the host from most other activity.

4.5.2.2 Serverless Workloads

FunctionBench workloads (Figures 4.4b and 4.4e) reveal distinct sharing patterns driven primarily by memory-management and filesystem structures. During *cold start*, all platforms exhibit elevated acquisition rates, with sharing consistently concentrated in memory-management and filesystem subsystems. Across workloads, the dominant objects are `zone`, journaling structures such as `journal_s`, and page-cache related objects such as `address_space`; model training, video processing, and CNN stress allocator and journaling paths more heavily, while RNN shows a narrower but still allocator-driven spike. Platform design shapes how this sharing manifests: *lxc* exposes a broad mix of host VFS and memory objects due to direct interaction with page-cache and filesystem metadata; *fc* narrows object diversity but concentrates peak acquisition on `zone`, `journal_s`, and scheduler structures such as runqueues during VM setup and virtio-backed I/O; and *gv* presents a broader but generally lower-intensity profile, as memory faults and file accesses still converge on host `zone` and journaling paths despite user-space mediation.

In *steady state*, workloads separate into two groups. Image and video processing maintain comparatively higher sustained sharing across platforms due to repeated file I/O and memory access, keeping `journal_s` and `zone` active. In contrast, model training and RNN

stabilize after initialization and become more compute-bound, with sharing largely confined to allocator structures such as `zone` and occasional `lruvec` activity. CNN and face detection occupy a middle ground, showing moderate sustained allocator and page-cache interaction. Platform differences persist in the steady state. *lxc* continues to expose a broader set of VFS and memory objects, reflecting direct host-kernel interaction. *fc* maintains a narrow-and-deep profile, concentrating sharing on allocator, journaling, and scheduler objects. *gv* spreads interaction across a moderate set of objects but generally at lower acquisition intensity.

Unlike microbenchmarks, serverless workloads spend much of their lifetime in cold-start and bursty execution phases [135], making transient sharing patterns central to isolation. Isolation is therefore shaped less by steady-state averages and more by how platform design amplifies or dampens short-lived object-level sharing. *lxc* exposes broad and consistently intense sharing, *gv* maintains lower-intensity behavior, while *fc* concentrates sharing into high-intensity cold-start bursts, creating focused interference despite stronger steady-state isolation. This highlights a trade-off between efficiency and isolation: aggressive cold starts improve utilization but increase object-level sharing during initialization.

4.5.2.3 Cloud Workloads

Cloud workloads shown in Figures 4.4c and 4.4f diverge most across platforms, reflecting how each platform handles sustained, resource-heavy execution. In *steady state*, *lxc* consistently shows both high lock counts and high acquisition rates across data analytics, data serving, graph analytics, and in-memory analytics, showing a broad and intense exposure to the host kernel, leaving these workloads highly vulnerable to object sharing from co-located workloads. *fc*, in contrast, maintains a narrower footprint with fewer locks but continues to exert high pressure on specific hot objects, notably in data analytics and in data serving. While it achieves negligible sharing with in-memory analytics and media streaming, its narrow but deep sharing profile in data-heavy tasks suggests that isolation is

concentrated rather than eliminated. *gv* occupies an intermediate position: it has moderate lock coverage but has lower rates, except in media streaming, surpassing *fc*.

As we previously observed for serverless workloads, this divergence is also visible in the diversity of kernel objects; *lxc* uses 43 unique locks, *gv* 25, and *fc* 15 across modes. *lxc* spans sharing across diverse subsystems: `uts_sem` (protects uts namespace), `proc_subdir_lock` (protects procfs task subdirectory lists), and `swap_lock` (protects global swap subsystem state), reflecting exposure to broad kernel resources. *gv* shows virtualization-specific locks like `free_vmap_area_lock` (protects vmap area allocation lists) and `folio_wait_table[i]` (protects folio waitqueues), while *fc* adds fewer unique locks but includes important shared locks like `journal->j_wait_commit`.

For cloud workloads, isolation is driven by sustained interaction with shared kernel services rather than short-lived phases, making steady-state sharing the primary concern for long-running co-location. Platform design determines whether this pressure is broadly distributed across many kernel objects, as in *lxc*, or concentrated on a small set of hot objects, as in *fc*, with *gv* occupying a middle ground by spreading access while keeping intensity lower. Hence, for cloud workloads, isolation is less about avoiding transient spikes and more about controlling long-term exposure to shared kernel state through platform design.

Summary: Across all workload categories, we find that the same workload can vary significantly in object usage across platforms, shown in both lock counts and acquisition rates. This implies that a workload's sensitivity to objects depends strongly on the platform it runs on. Platform design alters not only the intensity of object usage but also the type of sharing: *lxc* exposes the widest variety of objects, *gv* introduces new synchronization points in memory and filesystems, and *fc* narrows diversity but concentrates pressure on hotspots like journaling and memory. Thus, sharing is platform-specific, and selecting the right platform for a workload directly shapes its object-level interference profile.

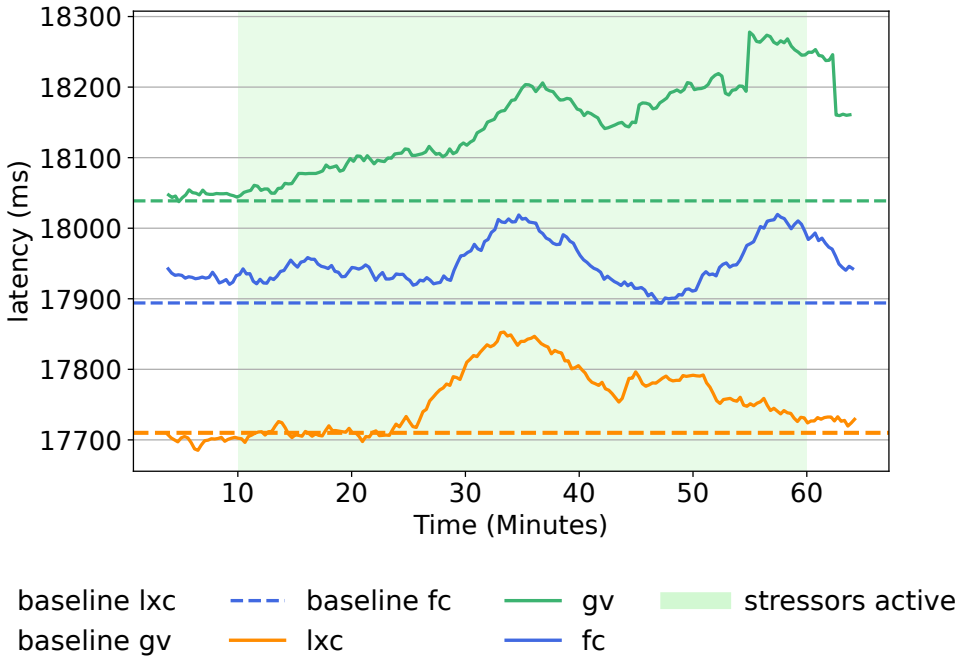


Figure 4.7: Performance impact on model training when run with varying stress-ng *stressors*.

4.5.3 Dominant Kernel Subsystems

Kernel object usage is spread widely across the Linux codebase. We plot the total of maximum average access rate (summed over the maximum rate for unique locks across modes for a given workload) for FunctionBench and cloud workloads in Figures 4.5 and 4.6, respectively. Our results show that across platforms, most acquisitions are concentrated in *mm* and *fs*, followed by *kernel*. Improving workload isolation, therefore, requires rethinking how memory and filesystem state are shared, rather than simply reducing exposure to less frequently exercised kernel subsystems.

4.5.4 Performance Isolation

To understand the impact of object sharing on performance interference, we run each FunctionBench workload for 70 minutes, launching a different stress-ng *stressor* every 10 minutes, starting at 10 minutes in sequence: *getdent* (10-20), *fstat* (20-30), *io* (30-40), *mem* (40-50), and *fork* (50-60). Figure 4.7 shows model training, where latency degrades as

stressors are introduced. *lxc* and *gv* experience the largest slowdown. *fc* is more resilient, but still has some performance dips. These results highlight that the degree of performance sensitivity is coupled to the breadth and focus of lock sharing exposed by each platform. We make similar observations for other FunctionBench workloads.

Summary: Object-level sharing translates into measurable performance interference under co-location, with severity depending on platform design. These results suggest that placement decisions should also account for hot shared kernel objects, such as memory-management or filesystem-journaling structures, in addition to CPU, memory, and I/O demand, to improve predictability.

4.6 Summary and Discussion

Breadth and depth capture different risks. Our sharing patterns vary along two axes: how many shared objects a workload accesses (breadth) and how heavily it accesses each object (depth). A broad footprint creates more possible points of interference with co-located workloads, while a narrow but deep footprint can create substantial interference when a neighbor accesses the same hot objects. Our platform results show that these dimensions can move independently: *fc* reduces the number of shared objects but concentrates activity on a few objects such as `&zone->lock`, trading breadth for depth rather than eliminating exposure. Neither dimension alone therefore determines the impact of sharing. Impact depends both on whether workloads overlap on the same objects and on how heavily those objects are exercised. Chapter 5 captures this distinction quantitatively: each shared object contributes an interference point, while its access rate and associated risk determine the contribution of that point to `EoR`. Furthermore, the two dimensions also differ in how they can be mitigated: because depth concentrates exposure on a few objects, addressing a single component (*e.g.*, splitting up a hot object or converting it to a per-CPU state) can remove much of the interference at once, while a broad footprint spreads interference

potential across many subsystems and offers no single point of intervention.

Platform design shifts where interference appears. Kernel-level object sharing is unavoidable across platforms, but platform design determines how it manifests. *lxc* exposes a broader set of kernel objects, *gv* introduces new synchronization points tied to its virtualization layer, and *fc* reduces lock diversity but heavily uses objects such as `&zone->lock` and journaling locks. While a core set of hot objects persists across all three, platform design determines which additional subsystems become exposed to sharing. Isolation, hence, is not achieved simply by narrowing object accesses; platform design must explicitly consider how new mechanisms shift sharing to different parts of the kernel.

Quantifying isolation beyond conventional wisdom. Although VMs are known to reduce shared state by interposing a guest kernel, this understanding is largely qualitative. Our study quantifies isolation by measuring how platforms differ in the breadth and intensity of shared kernel objects. Rather than treating isolation as binary, we show how platform design reshapes object-level sharing, enabling workload-specific comparisons and exposing trade-offs hidden by coarse abstractions like VM vs. container.

Mitigation requires targeting hot objects. Across all platforms, a small set of memory-management and journaling locks repeatedly emerges as shared bottlenecks. While some pressure can be reduced at the workload level (*e.g.*, disabling journaling for short-lived serverless tasks), stronger isolation requires privatizing or partitioning frequently contended kernel structures [84]. In particular, further partitioning allocator and journaling data structures could meaningfully reduce cross-workload interference. Focusing on these recurring hot objects is likely more effective than broad, system-wide redesigns.

Cold start magnifies sharing. Initialization paths show elevated lock usage, and virtualization layers amplify this. The implications extend beyond latency: cold starts intensify object-level sharing, making serverless workloads more vulnerable to interference during startup.

Object sharing matters for co-location. Our analysis shows that object-level sharing

impacts performance: co-located workloads often contend on a few hot kernel objects. Cloud schedulers account for CPU, memory, and I/O usage, but rarely consider interference through shared objects like memory zones or journaling locks. Accounting for object-level usage in placement decisions could therefore improve predictability and reduce interference — a question we take up in Chapter 5, where we formalize a workload's exposure to shared resources into a quantitative isolation metric.

4.7 Conclusions

In this chapter, we introduced and evaluated a new approach to understanding data isolation via shared kernel objects by analyzing kernel-level lock activity. Across workloads and isolation platforms, we found significant variation in object access patterns, with the file system and memory management subsystems emerging as the most frequent sources of sharing.

— 5 —

Out of the Woods, Into an Isolation Metric

> All models are wrong, but some are useful.
>
> *George E. P. Box*

This chapter formalizes isolation as a measurable quantity. The preceding chapters show that a workload's isolation depends on the kernel code and objects it exercises and on the neighbors it runs alongside; here we develop a framework that turns this dependence into a number. We define the key entities of a workload's environment — the host, the platform runtime, co-running workloads, and the resources they share — and propose a new isolation metric, *Expectation of Risk* (EoR), which quantifies the isolation a workload receives by combining interference magnitude with associated risk at each shared interference point.

The goal of the metric is to allow comparison of the relative isolation an executing workload gets under different environments and contexts, and to determine which environment offers *enough* isolation for a given workload. We evaluate EoR using shared kernel locks as a proof-of-concept resource, asking two questions: (a) does EoR correlate with observed performance interference, and (b) can EoR distinguish between platforms with different isolation properties?

Some contributions of this chapter are:

- A system model of an isolated workload's environment that identifies where interfer-

ence can occur and how severe it may be.

- The definition of EoR and an analysis of its properties, including how interference magnitude and risk are derived for different resource types.
- A proof-of-concept instantiation of EoR using kernel lock access rates and hold times, showing that EoR correlates with measured performance change under interference and is consistent with the relative isolation strength of the platforms we study.

The rest of this chapter is organized as follows. We present the system model of isolation (§5.1), define EoR and show how isolation can be broken down numerically (§5.2), analyze the properties of EoR (§5.3), discuss its scope and limitations (§5.4), evaluate EoR empirically (§5.5), and conclude (§5.6).

5.1 A System Model of Isolation

As stated (§1.4.3), we need knowledge of the workloads' environment to define and measure isolation[1]. Therefore, to quantify isolation, we first discuss the system model of the cloud environment. The model captures all the components (*e.g.*, isolation platform executing the workload) and contexts (*e.g.*, co-running workloads, resources accessed) required to formulate and define isolation. This model serves as a building block for the isolation metric, *Expectation of Risk* (§5.2), and provides a standard framework for formally defining the metric.

Table 5.1 summarizes all symbols used in the model. We describe each component and the environment below.

Host, H is a computer system or a physical server in a cloud cluster that runs multiple workloads simultaneously.

[1]Or as Michael Swift put it: "We cannot define isolation in isolation."

Symbol	Description
H	Host
$\mathtt{Platform}_{\mathtt{shared}}$	Shared HW and SW components of H
w	Workload
$\mathtt{Runtime}_w$	Isolation platform runtime executing w
s	An isolated workload running on H
R_s	Set of resources accessed by s
T_s	Set of all isolated workloads on H excluding s
E_s	Environment of s
$\mathcal{I}_s$	Interference resources of s with T_s

Table 5.1: Symbols used in the isolation system model.

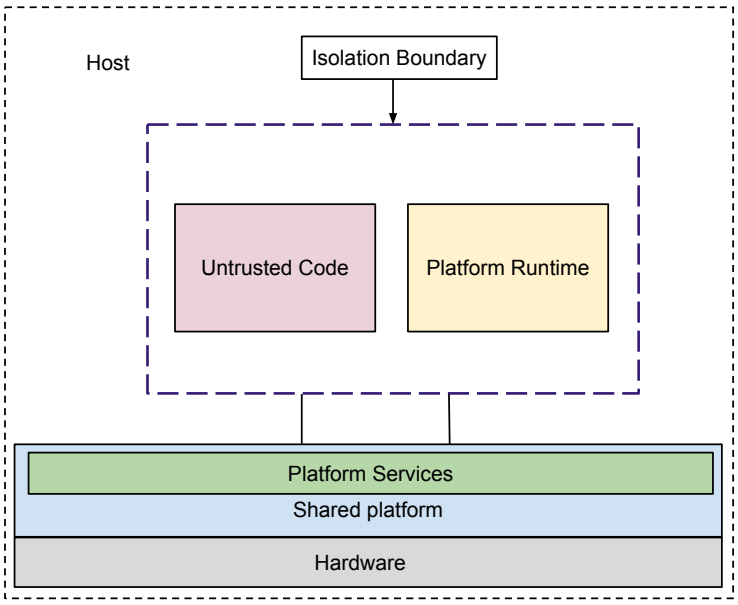


Figure 5.1: Components of an isolation platform.

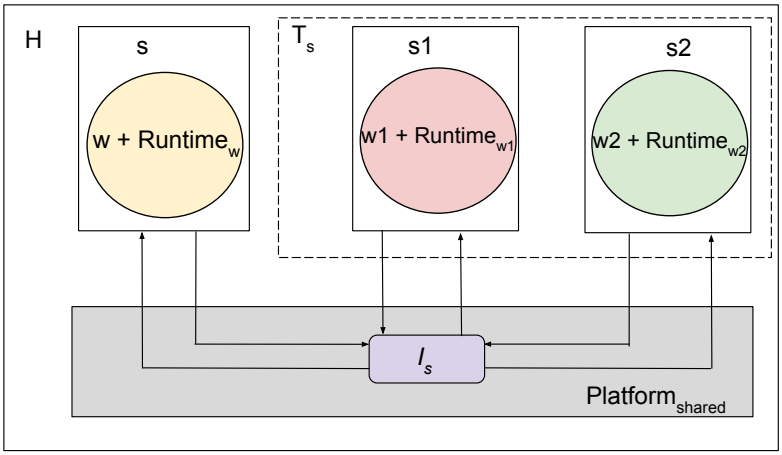


Figure 5.2: Environment, E_s of an isolated workload, s in the isolation system model.

Shared Platform, $\texttt{Platform}_{\texttt{shared}}$ consists of shared hardware and the set of processes running in the lower software layers (*e.g.,* host kernel) of H managing the system resources.

Isolation Platform is a set of processes running on a shared platform that provides an isolation boundary to a workload by leveraging services implemented in the shared platform, by implementing additional software layers above the shared platform, or both. We recognize two main components of isolation platforms, shown in Figure 5.1:

- *Platform Service*(s) is the set of software mechanisms that the shared platform implements, and the isolation platform leverages in creating the isolation boundary for a workload (*e.g.,* cgroups, KVM). This component is *shared* across workloads.
- *Platform Runtime* is the set of software layers the platform implements above the shared platform, in userspace, moving functionality out of the shared platform to lower the Trusted Computing Base (TCB) and limit sharing at the lower layers (*e.g.,* Sentry and Gofer in gVisor). This component is *private* to workloads; we define it formally below.

Formally, an isolation platform is a set of processes that uses different software layers to create an isolation boundary for a workload by restricting what the code above can do. We represent it as $\texttt{Platform}_{\texttt{isolated}} = \{\texttt{proc}\}$: for example, a gVisor container is represented as $\texttt{Platform}_{\texttt{gVisor}} = \{\texttt{sentry}, \texttt{gofer}, \texttt{containerd}, \texttt{runsc}, \texttt{kvm}\}$, a regular container (LXC) as $\texttt{Platform}_{\texttt{LXC}} = \{\texttt{containerd}, \texttt{runc}\}$, and a Firecracker microVM as $\texttt{Platform}_{\texttt{firecracker}} = \{\texttt{firecracker VMM}, \texttt{jailer}, \texttt{kvm}\}$.

Platform Runtimes are software layers implemented by an isolation platform (*e.g.,* LXC, gVisor, Firecracker) above the shared platform layer and are part of userspace. This is *private* to workloads and is represented as $\texttt{Runtime}_w$ for a workload, w.

Isolated Workload Instance, s, is a combination of w, and its platform runtime, $\texttt{Runtime}_w$ (*e.g.,* a DB server running in a gVisor container) running on H. As both $\texttt{Runtime}_w$ and w are made up of processes, s is also a set of processes:

$$s = w + \texttt{Runtime}_w$$

Resource. A resource within H is any physical (*e.g.,* CPU, memory, disks, devices) or virtual component (*e.g.,* files, network socket, locks, task queues) with limited availability used by s to execute a task.

Each s accesses a set of resources, R_s, during its execution. R_s is divided into private and shared resources. s's private resources (*e.g.,* private memory regions and open files) are exclusive to s and are not used by other s on H. Shared resources are accessible and globally available to all s running on H (*e.g.,* task queues, network devices, shared files). Hence,

$$R_s = R_{s,private} + R_{s,shared}$$

Environment. We define the environment for s executing a set of tasks (shown in Figure 5.2), $E_s = <R_s, Platform_{shared}, T_s>$ where R_s consists of all resources s uses, $Platform_{shared}$ is the shared platform, and $T_s = H \setminus \{s, Platform_{shared}\}$ is the set of all processes that exclude s and $Platform_{shared}$, *i.e.,* set of all isolated workloads excluding s.

Interference Resources. The shared resources that s has in common with T_s [35] are:

$$I_s = R_{s,shared} \cap (R_{s_1,shared} \cup R_{s_2,shared} ... \cup R_{s_n,shared})$$

In E_s, where system resources are shared, concurrent s and T_s can interfere with each other at different resources (*e.g.,* shared structures, devices, concurrent memory accesses). These resources are called *interference resources* (I). I_s captures the interference resources of s with T_s in E_s. Note that the pairwise sharing of resources is non-transitive, *i.e.,*

$$x \in (R_{s,shared} \cap R_{s_1,shared}), y \in (R_{s,shared} \cap R_{s_2,shared})$$
$$\not\Rightarrow x, y \in (R_{s_1,shared} \cap R_{s_2,shared})$$

Capability. The capability of a workload is a set of operations, read-only (*ro*) and read-write (*rw*) on resources during its execution. Based on the capability, a workload can access

or modify these resources. Every resource, r_i (the i^{th} resource, where i ranges from 1 to m, the total available resources for s) in R_s is associated with a capability, *i.e.,* the operations are per resource. Hence,

$$R_s = \{r_{1_{(private,ro)}}, r_{2_{(shared,ro)}}, r_{3_{(private,rw)}}, ..., r_{m_{(shared,rw)}}\}$$

Interference Context. We identify two contexts for isolation.

- *Benign context* happens when multiple workloads are running in the system with no intention of affecting others, but can unknowingly affect one another, *i.e.,* all s on H are benign. Here, R_s is the set of resources referenced by s with the *actual* capability — the resources a correct program accesses.
- *Malicious context* happens when multiple workloads are running in the system with at least one workload intentionally trying to impact performance, modify data, or change the control flow of all or at least one other workload, *i.e.,* at least one s on H is malicious. R_s in this context is *all accessible resources* with *whatever* capabilities are possible by s.

Our formulation of isolation does not incorporate any particular contexts or threat models, assuming that interference in any context can potentially lead to undesirable outcomes.

Expected Behavior. Suppose s executes a sequence of instructions to finish running in a specific execution time on H with a set of resources where no other isolated workload in T_s is running. This is the *expected behavior* of s without interference from T_s.

Interference. In environment E_s, workload s can be interfered with by other workloads T_s if interference resources $\mathcal{I}_s \neq \emptyset$: a nonempty $\mathcal{I}_s$ establishes the potential for interference, not its occurrence. Conversely, $\mathcal{I}_s = \emptyset$ implies no interference is possible via shared resources. Whether and how strongly this potential is depends on how s and T_s exercise the shared resources — which is what EoR quantifies.

Isolation is the degree of how much s is insulated from T_s, *i.e.*, what is the *impact* on the expected behavior of s if s is interfered with by the presence of T_s. s is *fully isolated* if its *expected behavior does not change* (*e.g.*, visible changes such as performance degradation, change in control flow, or invisible changes such as data leakage, and may be deterministic or nondeterministic) when system resources are shared.

Relation to non-interference. Our notion of full isolation is closely related to the classical security property of *non-interference* [76], which requires that one domain's actions do not affect another domain's outputs or behavior. Adapting this idea to our system model, we treat the observable behavior of s as including execution time, control-flow outcomes, and data. With respect to the modeled resource dimension, $I_s = \emptyset$ implies non-interference: there is no shared resource through which T_s can influence s. Without shared resources, s and the workloads in T_s also cannot observe one another's resource usage, ruling out information leakage. Conversely, *non-isolation* is an observable departure of s from its expected behavior attributable to T_s. A nonempty I_s indicates only the potential for such a change; its realized magnitude depends on how s and T_s use the resources in I_s. Isolation is therefore a matter of degree, motivating its quantification.

Scope. Our formulation captures only interference that occurs through a modeled resource. It includes any benign or malicious interference mediated by a shared resource represented in I_s, but excludes channels not represented in the model. In particular, the empirical `EoR` instantiations in this dissertation quantify performance interference through shared host-kernel state; they do not verify information-flow security or confidentiality. Consequently, a workload may score low on our lock-based `EoR` yet remain exposed through shared resources outside this instantiation — for example, microarchitectural state such as caches, which may be exploited as a side channel. Such resources are not outside the formulation — the model admits them, and §5.4 sketches possible `EoR` instantiations for other resource types — but quantifying them requires a separate analysis with its own risk definition.

We use all these components and the environment of the model to define the metric

next. We note that these definitions can apply to just a single resource (*e.g.,* files), categories of resources (OS objects), or all resources (OS objects and hardware resources).

5.2 Breaking down Isolation, Numerically

Expectation of Risk (EoR) measures the likelihood of other workloads T_s interfering with target workload s via resources $\mathcal{I}_s$, and how badly it impacts s (*e.g.,* amount of performance degradation or data leakage), *i.e.,* the risk of such interference. Sharing resources creates opportunities for interference, increasing the chance of interference with more sharing. The risk of interference is modeled within a specific context: *What type* (e.g., *sensitive vs. non-sensitive*) *of shared resources are accessed and how* (*ro vs. rw*)*? What is the resource access pattern of the workload* (e.g., *stable vs. variable*)*? Is the access to the shared resource blocking or non-blocking?* Building on these factors, we present some new concepts and principles for defining the metric.

Unit of Interference. Workloads can interfere with each other at multiple interference resources ($\mathcal{I}$), each resource impacting isolation differently. Moreover, there are different ways a workload can be interfered with via these interference resources (*e.g.,* performance, security, data). For instance, if we want to measure contention due to performance, we would quantify interference using time and rate. Likewise, for security-related interference, the unit might be the likelihood of unauthorized access to sensitive data by finding the number of exposed system calls, shared resources *etc.*

We abstract these different ways to measure interference via an interference resource into a unit of interference, which we call an *isomphere* (*iso*). An *iso* is an alias for different possible measurement units used to quantify interference. It may represent a rate, volume (amount/size), or a combination of both, depending on the resource and type of impact.

Definition 5.1. *Unit of Interference, called isomphere* (*iso*) *is a quantity that measures the opportunities of interference for a workload that can incur through an interference resource.*

Interference Magnitude. The unit of interference, $\mathtt{iso}$, specifies how interference is measured, whereas the interference magnitude specifies the value of that measurement. This magnitude need not represent an absolute quantity; its form depends on the interference dimension and the risk being modeled.

Definition 5.2. *Interference magnitude quantifies how much a workload is interfered with through a shared resource. It may be captured as a probability* (e.g., *likelihood of a workload experiencing interference through a shared resource*), *absolute amount* (e.g., *duration or rate*), *or any context-aware quantity* (*size of sensitive data being shared*). *It depends on the nature of the risk being modeled, the data on hand* (e.g., *granularity, accuracy, and type of data*) *and the measurement capability* (e.g., *tools for tracing and logs*).

The choice of $\mathtt{iso}$ determines how $\mathtt{EoR}$ is interpreted: computed over rates, $\mathtt{EoR}$ reads as exposure per unit time; over probabilities, as the likelihood of interference, weighted by its risk; over volumes, as expected amount of data at risk. The metric remains comparative in every case, but $\mathtt{EoR}$ values are only comparable when computed with the same $\mathtt{iso}$.

To quantify performance interference, the interference magnitude can be calculated using $\mathtt{iso}$ (rate, time), capturing how often or how much a workload is slowed down due to a particular shared resource. In the context of security, such as data leakage, the interference magnitude may correspond to $\mathtt{iso}$ (volume), the amount of sensitive data being exposed or shared. Directly measuring interference between workloads would require observing each workload's execution both with and without its neighbors. Instead, we profile the resource usage of each workload independently and infer interference at runtime from the resources their profiles share. Interference magnitude varies under different execution environments and contexts. We express the interference magnitude using a function over resource usage:

$$\mathtt{mag} = \phi_{\mathtt{iso}}(\mathcal{I}_s, \mathtt{R}_s)$$

where $\phi_{\mathtt{iso}}$ is the magnitude function over unit of interference, $\mathtt{iso}$ (rate, time, probability,

size *etc.*). Intuitively, if $\mathcal{I}_s = \emptyset$, then $\mathtt{mag} = 0$.

Associated Risk. Not every interference resource has the same impact on isolation; some interference resources can drastically lower isolation, whereas some have little to no effect on isolation. For example, multiple workloads with read-only (*ro*) operations accessing shared data protected by reader-writer spinlocks will be less vulnerable than workloads with read-write (*rw*) operations that heavily use kernel services modifying many shared resources.

Definition 5.3. *Associated Risk is a metric that quantifies the risk associated with each interference resource. The higher the risk of accessing an interference resource, the more vulnerable and less isolated a workload is through that interference resource.*

Associated Risk abstracts the severity of interference through an interference resource. We express the associated risk along three *context-dependent* dimensions: (1) access type (*rw* and *ro*), (2) access sensitivity (sensitive vs. non-sensitive resource), and (3) nature of access (blocking vs. non-blocking). For access type, write operations can alter shared state and impact the behavior of other workloads. Read-only operations do not impact the state of other workloads, but can induce delay and alter performance. Sensitivity indicates the value of the information protected – how bad is it if something leaks or changes? Blocking access (*e.g.*, mutexes, synchronous I/O) can incur delay and increase performance interference compared to non-blocking access.

For performance, risk may be the expected slowdown due to a shared resource that is blocking. In the case of information leakage for security, if we have knowledge of the amount ($\mathtt{iso}$) of sensitive data to be accessed, we can quantify risk as 1 for such accesses and 0 for public data. This will measure the total leakage assuming all sensitive accesses are bad (worst-case scenario). Deriving accurate risk values is non-trivial and may require domain knowledge, similar to estimating damage-to-effort ratios in attack surface metrics [104].

Expectation of Risk. We measure isolation by calculating the overall *Expectation of Risk*

(EoR).

Definition 5.4. *Expectation of Risk is a weighted sum of all the possible associated risk values that a workload running in a given environment can incur through all interference resources, each value weighted by its interference magnitude. This metric is abstract and comparative, making it suitable for comparing isolation between workloads/platforms/environments. Therefore,*

$$\mathtt{EoR} = \sum_{i \in \mathcal{I}} \mathtt{mag}(i) \cdot \mathtt{r}(i) \tag{5.1}$$

Here, i denotes an interference resource in the set $\mathcal{I}_s$, $\mathtt{mag}(i)$ is the interference magnitude through *i*, and r_i is the risk associated with interference at *i*. EoR can be calculated independently along different resources (*e.g.,* CPU, memory, caches), or kernel subsystems (*e.g.,* filesystems, networking) to measure different dimensions of isolation (*e.g.,* performance, security).

Isolation. Workload s is fully isolated if its $\mathtt{EoR}_s = 0$, indicating no risk and complete isolation, *i.e.,* the *expected behavior* of s remains unchanged. Moreover, $\mathcal{I}_s = \emptyset$ also implies $\mathtt{EoR}_s = 0$ and complete isolation.

When $\mathtt{EoR} > 0$, it implies s is vulnerable to some risk due to interference, potentially leading to *change in the expected behavior* of s.

5.3 Properties of Expectation of Risk

We see several valuable use cases for the expectation of risk metric:

Isolation on the same platform. We can run multiple workloads in an isolated platform and use the metric to observe how they impact each other's expected behavior. This can help schedule workloads on a platform that has the least impact on each other. Thus, cloud providers can use it to run workloads together that do not interfere with each other by choosing those with the lowest expectation of risk when running together.

Claim 5.5. *Given a system, S and an environment* $E = \langle R, \texttt{Platform}_{\texttt{shared}}, T \rangle$*, workloads, A, B and C, running in separate isolation platforms where* $\texttt{Platform}_{\texttt{isolated,A}} = \texttt{Platform}_{\texttt{isolated,B}} = \texttt{Platform}_{\texttt{isolated,C}}$ (e.g., *all workloads running in a VM*). *If* $\texttt{Exp}[R_B] > \texttt{Exp}[R_A] > \texttt{Exp}[R_C]$*, it implies that A is more isolated in the presence of B than C in E when run in the same isolation platforms.*

Isolation across platforms. The same workload can be executed on two different isolation platforms, and the metric can be used to compare the level of isolation across platforms. This will help developers in deciding whether the added isolation of the platform is worth the increased cost (in performance of dollars). For example, if a workload has the same EoR when run as an OS process or on dedicated hardware, there is little value in the added cost of dedicated hardware.

Claim 5.6. *Given a system, S, and an environment* $E = \langle R, \texttt{Platform}_{\texttt{shared}}, T \rangle$*, workloads, A and B, running in isolation platform,* p_1*. Let* $\texttt{Exp}[R_A]_{p_1}$ *represent the expectation of risk of A when run in parallel with B. Let workloads, A and B, now run in isolation on platforms,* p_2 *and* p_1*, respectively. Let* $\texttt{Exp}[R_A]_{p_2}$ *be the expectation of risk for A in this scenario. If* $\texttt{Exp}[R_A]_{p_1} > \texttt{Exp}[R_A]_{p_2}$*, it implies that A is more isolated in the presence of B (isolated in* p_1*) when running in the isolation platform* p_2 *in E, compared to isolation platform* p_1*.*

Impacts of different interference points on isolation. Different interference points impact isolation differently. Comparing per-point EoR contributions identifies which shared resources dominate a workload's exposure — on one platform across environments, or across platforms for the same workload. This identifies where stronger isolation or a different platform would reduce risk the most.

Claim 5.7. *Given a system, S, and an environment* $E = \langle R, \texttt{Platform}_{\texttt{shared}}, T \rangle$*, isolated workloads, A and B running in parallel. If* $\alpha = \{i_1, i_2, ..., i_n\}$ *and* $\beta = \{r_{i_1}, r_{i_2}, ..., r_{i_n}\}$ *represent the set of*

Resource	Isolation	EoR	Magnitude	Risk	Data needed
Data Structures	Performance	Slowdown via shared objects	Rate	Average delay due to contention	Traces of objects shared during execution
	Security	Risk of data leakage and corruption	Size of shared data	1 (sensitive/write) 0 (insensitive/read)	
Cache	Performance	Slowdown due to cache misses	Cache miss rate	Average delay due to misses	Perf counter for cache misses
	Security	Risk for side channel leakage via shared cache	Probability of overlapping access to a cache line with other workload	1 (sensitive) 0 (insensitive)	Cache access traces, timing analysis, knowledge of shared cache sets
Memory	Performance	Slowdown due to shared bandwidth	Bandwidth	Delay	Bandwidth usage, queuing delay at memory controller
	Security	Risk of sensitive memory access	Probability of co-location with sensitive memory	1 (sensitive) 0 (insensitive)	Memory allocation, access traces, tags for sensitive/ insensitive memory
Storage	Performance	Slowdown due to I/O interference	Bandwidth	I/O throughput degradation	I/O stats, request latency
	Security	Access to sensitive shared files	Probability of access to sensitive files	1 (sensitive) 0 (insensitive)	File access logs, sensitivity labels

Table 5.2: Examples for calculating EoR for different resources.

interference points and the associated risk values, r_i *for each i respectively such that,*

$$r_{i_1} \geqslant r_{i_2} \geqslant r_{i_3} ... \geqslant r_{i_n}$$

This implies that the higher the value of r_i*, the greater the chances of* i *lowering the isolation for A and B when run in parallel.*

5.4 Discussion

We now discuss some open questions regarding the practical application of this metric. We hope that applying this metric in calculating isolation along performance, security, or any other dimension will pave the way for new research.

Generality of the metric. We believe the proposed metric is inherently generic and is applicable for measuring different types of isolation, including performance, security, data, and more. Based on how risk is defined, the metric can be adapted in different scenarios. EoR measures a single class of risk, but it can apply to many types – contention for CPU, contention for memory or access to a device, likelihood of a compromise, *etc.* In Table 5.2 we show how we can potentially calculate EoR for different resources. Each resource requires

a separate analysis to quantify EoR. We note that in some cases, EoR may not be accurately computable (*e.g.*, attacks via kernel bugs) due to a lack of measurement data or other constraints. In such cases, coarser proxies, such as the kernel code footprint measured in Chapter 3, can substitute for finer-grained signals. Further, one cannot directly compare EoR for different risks without human input on the relative magnitudes of different risks. The goal of EoR is not to achieve absolute precision, but rather to approximate the likelihood and impact of interference due to shared resources.

EoR use cases. Having a metric to estimate isolation for a workload has several benefits from different users' perspectives. Cloud providers can use this metric while making placement decisions for a workload, as can developers when choosing how to deploy components of an application. Platform developers can compare design alternatives such as partitioning vs. sharing resources and evaluate isolation trade-offs. A clear metric can also facilitate designs of novel isolation techniques, scoring better on the metric, leading to a new path for innovation in the future. OS developers can apply the metric in identifying high sharing and contention points in different kernel subsystems, enabling them to better optimize the usage of such shared resources in the system (coarse-grained vs. fine-grained sharing).

Isolation vs attack surface metric. EoR complements the attack surface metric [104] and the two can be used together. The attack surface metric quantifies how exposed a system is to potential attacks, *i.e.*, how *attackable* a system is, by counting attack vectors such as system calls or open ports; EoR captures the *impact of interference* in the presence of co-running workloads through shared resources. Both metrics serve different but complementary purposes and can be used to compare the isolation guarantees of platforms.

Symbol	Description
C_l	Total lock count for lock, l
τ_w	Total execution time of w (sec)
λ_w^l	Access rate of w for lock, l (C_l/τ_w)
β_w^l	Average hold time for lock, l by w (sec)
D_w^l	Duty Cycle ($\lambda_w^l \times \beta_w^l$)

Table 5.3: Units used in calculating EoR.

5.5 EoR Empirical Evaluation

In this section, we evaluate EoR as a measure of performance isolation using shared kernel locks as the resource for our proof of concept. We use shared kernel locks as a proxy for sharing, as they are used to protect many shared resources in the kernel. Hence, by measuring lock behavior we can measure resource sharing in the system. We use kernel lock-sharing data collected from microbenchmarks and FunctionBench workloads (Chapter 4) for our evaluation. For each workload s in environment E_s (as illustrated in Figure 5.2) across different isolation platforms, we record all accesses to kernel locks and lock hold times.

As stated (§5.2), based on the risk being modeled, we need to calculate the interference magnitude and associated risk for EoR. We calculate the EoR on performance for workload s (victim) in environment E_s as shown in Figure 5.2. In our evaluation, we have one other workload in T_s, which we call a stressor, the competing workload acting as a noisy neighbor (we run two simultaneous workloads in H), to stimulate interference.

Our evaluation goals are: (a) Does EoR correlate with observed performance interference? (b) Can EoR distinguish between platforms with different isolation properties?

5.5.1 Identifying Interference Magnitude and Risk

Units of Interference. We use time and rate as our units of interference as described in Table 5.3.

Interference Magnitude. We express the interference magnitude as the rate of object access

for our scenario. Intuitively, interference occurs when s tries to acquire a shared lock, l in the same window when $stressor$ is holding l. That is, how often does s get slowed down by $stressor$? Therefore, we calculate the rate of interference (roi) via l as the interference magnitude, calculated using symbols described in Table 5.3 as: $roi_l = \lambda_s^l \times D_{stressor}^l$.

Risk. The risk to s in this scenario is the delay incurred when s attempts to acquire l while it is held by $stressor$. On average, this delay is approximated by half the hold time of l by $stressor$ *i.e.*, $r_l = \beta_{stressor}^l/2$.

We also distinguish between readers and writers when calculating interference magnitude and risk for reader-writer locks (*e.g.*, `rwlock`). For readers, we consider blocking only from writers; for writers, we consider blocking from both readers and writers.

For a **reader lock** l_{read}:

$roi_{l_{read}} = \lambda_s^{l_{read}} \times D_{stressor}^{l_{write}}, \quad r_{l_{read}} = \beta_{stressor}^{l_{write}}/2$

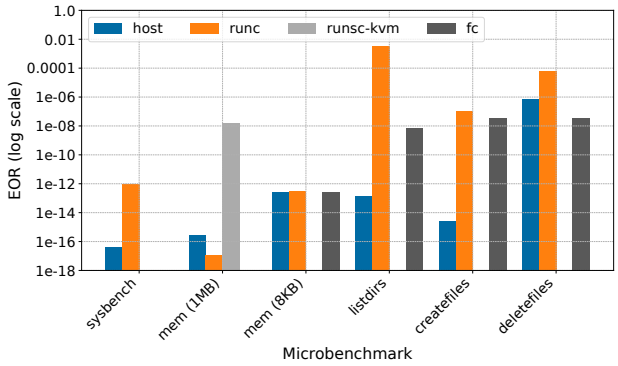
If there is no stressor holding a write lock on l, then the EoR for l_{read} is 0.

For a **writer lock** l_{write}:

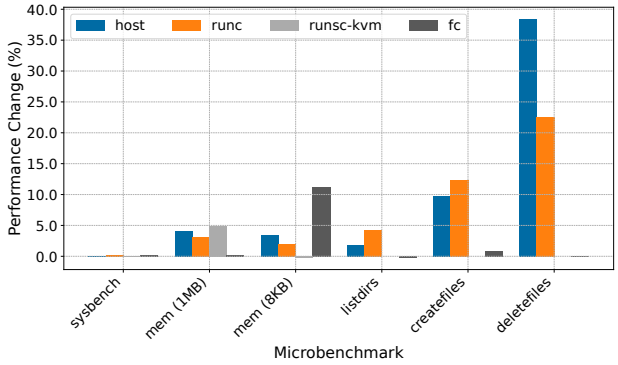
$roi_{l_{write}} = \lambda_s^{l_{write}} \times (D_{stressor}^{l_{write}} + D_{stressor}^{l_{read}}), \quad r_{l_{write}} = (\beta_{stressor}^{l_{write}} + \beta_{stressor}^{l_{read}})\ /\ 2$

EoR. From 5.1, we calculate the total EoR via all shared locks as: $EoR = \sum_{l=1}^{l=n} roi_l \cdot r_l$.

Our goal is to evaluate whether EoR accurately captures performance interference and compares platforms' isolation levels. To test, we design two interference scenarios: (1) when s and $stressor$ are the same workload (symmetrical), and (2) when s and $stressor$ are different (asymmetrical). For the symmetrical scenario, we use a series of microbenchmarks stressing particular subsystems. We pair each of six FunctionBench workloads with five different stress-ng stressors (a total of 30 workload–stressor combinations) for the asymmetrical scenario. Our core intuition is that when two workloads heavily utilize the same kernel subsystem (*e.g.*, file system), they are more likely to interfere with each other, reflected in higher EoR values (symmetrical). While we expect less interference with asymmetrical as they use different resources. We describe all the workloads below.



(a) EoR.



(b) Performance change. A positive change reflects degradation.

Figure 5.3: EoR and performance change for symmetrical workloads.

Microbenchmarks	**FunctionBench**	**stress-ng**
CPU, memory, file_metadata (Filebench)	image, video processing, model training, cnn, rnn, face detection	fork, vm, hdd, fstat, getdent

For each scenario, we (1) compute the Pearson correlation between EoR and performance slowdown for s with the *stressor* and (2) compare EoR values for s across platforms. In all graphs, the platforms, host, Linux containers, gVisor, and Firecracker are labeled as *host*, *runc*, *runsc-kvm*, and *fc* respectively.

5.5.1.1 Symmetrical Workloads

We show the Pearson's correlation between performance change and EoR for all symmetrical workloads on all platforms below (gVisor does not support disabling ASLR, a requirement for Filebench, so we exclude it from file metadata).

CPU	memory	file_metadata
0.01	0.58	0.91

We observe a very low correlation with CPU, implying that compute-heavy workloads do not interact much with shared kernel resources. Figure 5.3 shows negligible performance degradation and EoR value across platforms. The memory microbenchmark reveals a moderate correlation, indicating some interference from shared kernel resources. We observe a low EoR for most platforms, reflected by a low to moderate performance degradation by most runtimes (typically under 5%).

We observe the highest correlation for file metadata workloads. Both *host* and *runc* show moderate to high performance degradation for these workloads, with *runc* having the highest EoR value. *fc* shows no performance degradation despite a positive EoR value, suggesting that performance interference may arise from other sources in the system. While we minimize noise by reducing background system activity and enabling hardware-level isolation (*e.g.*, using Cache Allocation Technology (CAT)), we may still observe some interference due to factors that cannot be entirely eliminated.

Overall, these results indicate that EoR correlates with performance interference for workloads that interact heavily with the host kernel (notably file metadata). EoR for compute-heavy workloads with minimal interaction with the kernel shows 0 or near 0 EoR value. We also observe that EoR values are generally consistent with isolation strength across platforms, *host*, and *runc*, with a consistently higher value and a lower EoR for *fc*.

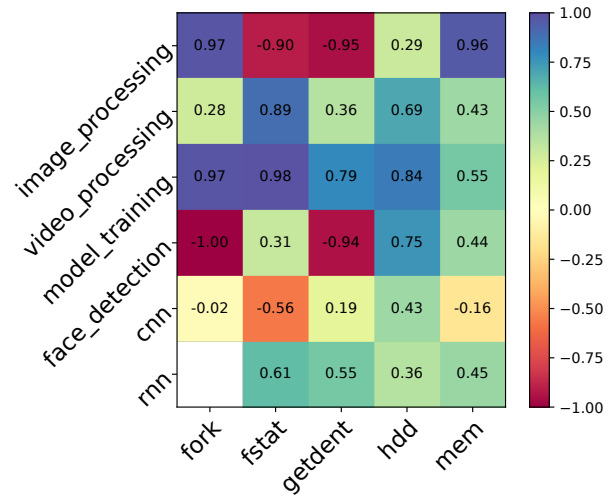
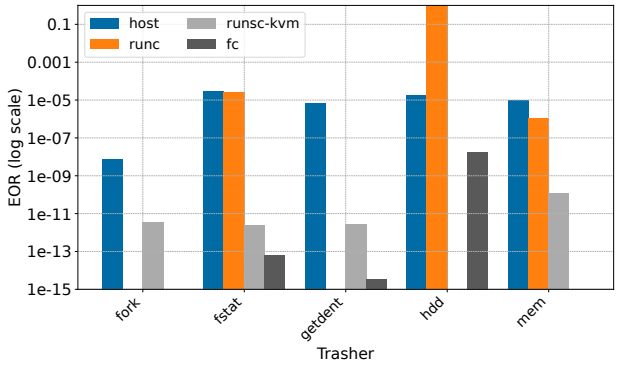
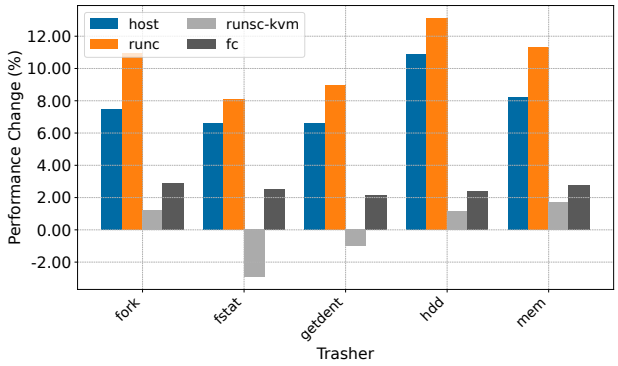


Figure 5.4: Pearson correlation for each FunctionBench (`s`) and stress-ng (`stressor`) pair.



(a) EoR.



(b) Performance change.

Figure 5.5: EoR and performance change for video processing.

5.5.1.2 Asymmetrical Workloads

We calculate the Pearson correlation using EoR and performance change for each workload-stressor pair across platforms as shown in Figure 5.4. Since `s` and `stressor` use different kernel resources, we expect lower EoR for this scenario, reflecting low interference. We observe varying levels of shared kernel resource sensitivity to performance interference across FunctionBench workloads as they are stressed across different resources (correlation for `rnn-fork` pair has no value as `fork` consistently has an EoR value of 0 across all pairs). Negative values imply that higher EoR is associated with improved performance, suggesting that the observed interference (*e.g.,* lock overlap) may not lie on the workload's critical path, and performance interference observed (if any) comes from other sources in the system. Altogether, we observe a strong correlation across pairs, validating EoR's utility as an interference indicator.

We observe that *fc* and *runsc-kvm* consistently have lower EoR values compared to *host* and *runc*, further showing EoR's comparative nature across platforms. This trend holds across most workloads and we show this for video processing in Figure 5.5. These results highlight our key evaluation goal *i.e.,* EoR's ability to compare platforms based on their isolation guarantees. The lower EoR values for *fc* and *runsc-kvm* align with their use of stronger isolation designs (*e.g.,* use of microVMs and a user-space kernel) by moving functionality away from the shared kernel, validating EoR's comparative ability.

5.6 Conclusion

To measure a workload's isolation in the cloud environment, we formulated and defined an isolation metric, Expectation of Risk (EoR). Our evaluation using kernel locks as a proxy for shared resources shows that EoR can effectively capture performance interference and compare platform isolation.

— 6 —

Isolation in the Age of Agents: An EoR Case Study

> The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.
>
> *Ada Lovelace*

This chapter applies EoR to an emerging class of workloads for which the isolation selection problem is especially acute: AI agents. Agents combine models for reasoning with tools for action [92], and the tool layer is where an agent's decisions become resource-consuming execution – reading and writing files, spawning processes, executing code, and issuing network requests – making tool execution a major point of contact with shared OS resources in multi-tenant environments [153, 48]. Because an agent's tool usage is heterogeneous, and its exact invocation sequence is not known in advance, deployments today rely on one-size-fits-all isolation choices for a diverse and changing toolset.

We characterize the resource usage and interference behavior of agent tasks under co-location on Linux containers, gVisor, and Firecracker, and instantiate EoR for shared host-kernel exposure using host-level syscall profiles and the shared kernel locks identified in Chapter 4, evaluating whether tool-call behavior can be used to estimate the isolation different platforms provide to agent tasks.

Some highlights of our findings are:

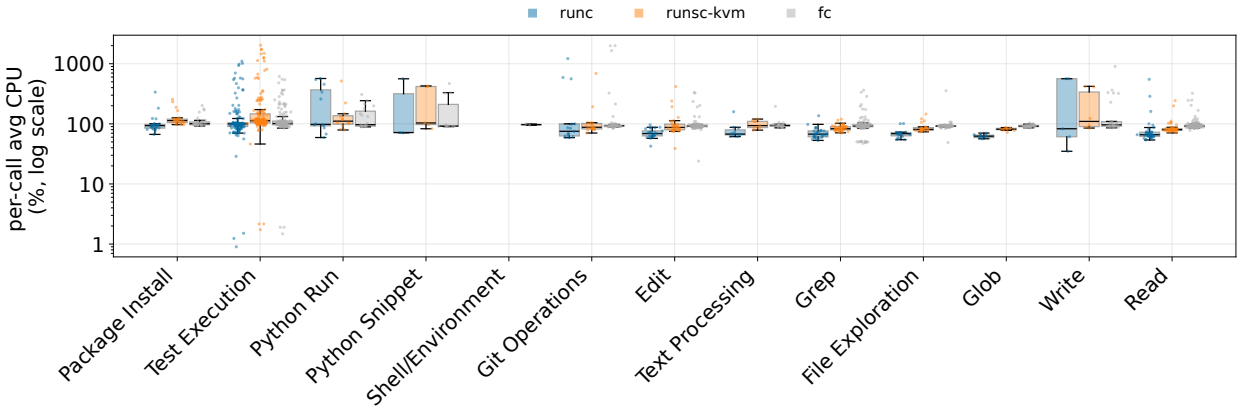
- Agent tasks experience substantial slowdown under co-location with $N = 10$ neighboring workloads: the median slowdown is approximately 64% on Linux containers, 25% on gVisor, and below 20% on Firecracker.
- Slowdown is heterogeneous across tasks, particularly on containers, meaning a fixed platform-level isolation choice can over-provision low-risk tasks or under-protect high-risk ones.
- Task-level EoR correlates with measured co-run slowdown pooled across platforms (Spearman $\rho = 0.67$), capturing the cross-platform differences in shared host-kernel exposure and providing a first step toward tool-aware runtime selection.

The rest of this chapter is organized as follows. We motivate tool-aware isolation (§6.1), describe our instantiation of EoR for agent tasks (§6.2), present our empirical evaluation methodology (§6.3) and results (§6.4), and discuss limitations and future work (§6.5).

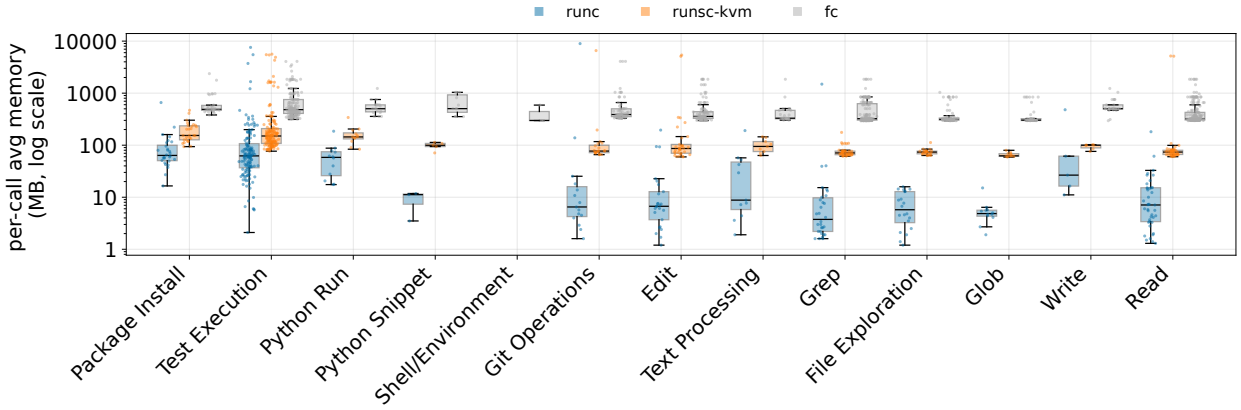
6.1 Motivation: Tool-Aware Isolation

Agent tool calls are the system-facing actions through which an agent accesses OS resources. A single task may invoke many tools, and each tool can stress different parts of the system: file-search tools interact with filesystem metadata, test commands spawn processes and consume CPU and memory, and package managers access filesystems, networks, package caches, and build tools. However, tool behavior alone does not determine isolation: the same logical tool may expose different amounts of shared host-kernel state depending on whether it runs in a container, secure container, or microVM.

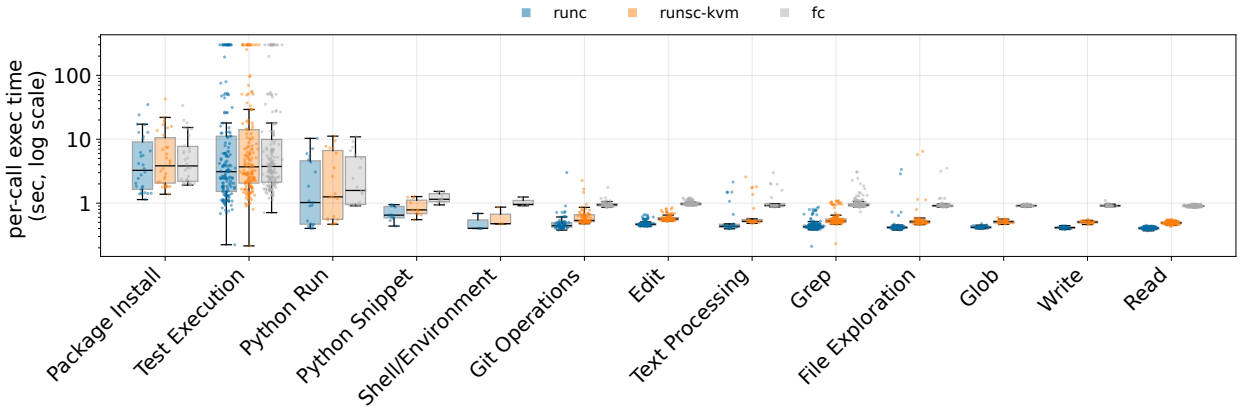
We replay Haiku tool-call traces from AgentCgroup [153], collected on SWE-rebench software-engineering tasks [43]. These tasks require agents to interact with real codebases



(a) Average CPU utilization per tool call. >100% = tool used multiple cores in parallel and <100% = time blocked/waiting off-CPU.



(b) Average memory usage per tool call.



(c) Execution-time distribution per tool call.

Figure 6.1: Resource utilization and execution-time distribution per tool category across isolation platforms.

by inspecting files, modifying code, running commands, and adapting based on execution feedback. Figure 6.1 shows the CPU utilization, memory usage, and execution-time distribution of all tool calls across all Haiku tasks on each runtime. We observe that tool behavior is highly heterogeneous: test execution and Python-based tools show higher CPU use, package installation tends to have larger memory and execution-time costs, and simple reads, edits, and text processing are comparatively lightweight. The same logical tool can also behave differently across platforms; for example, Firecracker (*fc*) often incurs a larger memory footprint due to the microVM boundary, while *runc* appears cheaper because it shares the host kernel directly.

This variation creates an opportunity for *tool-aware* platform selection: different tool calls may require different isolation boundaries depending on their resource behavior and shared host state exposure. A lightweight container may be sufficient for short, low-resource tools such as simple file reads or edits, while stronger isolation may be useful for tools that execute tests, spawn processes, install packages, or interact heavily with the filesystem and network. However, resource usage alone does not tell us how much isolation a runtime provides. A platform may consume more CPU or memory because it adds isolation layers, while exposing less host-shared kernel state. Conversely, a low-overhead runtime may appear efficient while exposing more shared host resources. Hence, choosing an isolation platform for agentic tools requires reasoning not only about resource cost, but also about the *shared system state* exposed by each runtime.

This motivates our research question: *Can we use the tool behavior within an agent task to estimate which platform provides sufficient isolation?* In this chapter, we take a first step toward tool-aware isolation selection by estimating the shared host-kernel state exposed by an agent task under different platforms. We use host-level syscall profiles as a lightweight signal of how the task uses the host kernel, map those profiles to shared kernel lock activity to compute task-level EoR, and use EoR to compare the isolation provided by different execution environments. Such a signal can later be combined with a workload's

performance and efficiency goals to make better placement decisions.

6.2 Instantiating EoR for Agent Tasks

Chapter 5 defined Expectation of Risk (EoR) as a combination of interference magnitude and risk over a workload's interference resources. Here we instantiate that model for agentic systems. The workload w is a tool invocation or a sequence of tool invocations issued by an agent, executed under a runtime that enforces an isolation boundary; the shared resources reachable by both the isolated workload s and its neighbors define the interference resources $\mathcal{I}_s$. For a task, we compute EoR as a weighted sum over these resources:

$$\mathrm{EoR}(s) = \sum_{i \in \mathcal{I}_s} \mathrm{mag}(i) \cdot r(i), \tag{6.1}$$

where $\mathrm{mag}(i)$ is the magnitude of the interference, capturing how much the workload uses the resource i, and $r(i)$ is the associated risk, capturing the severity of the interference through that resource.

As stated earlier, the metric is resource-specific: different isolation questions may instantiate $\mathcal{I}_s$, mag, and r differently. In this chapter, we target performance interference from host-shared kernel state. An agent task reaches the host kernel through system calls, and each system call exercises kernel code paths that acquire locks protecting shared kernel objects (Chapter 4); a task's syscalls therefore determine which shared kernel state its execution touches, and how heavily; we call this *shared-state exposure*. We estimate this exposure by profiling the host-level syscalls each tool invocation issues and mapping them to the shared host-kernel locks those syscalls acquire, which serve as the interference resources. This produces a task-level estimate of host-state exposure under each isolation runtime.

6.3 Empirical Evaluation Methodology

We evaluate whether agent tool-call behavior can be used to estimate the isolation provided by different isolation platforms/runtimes. Our evaluation uses tool-call traces from Haiku agent executions collected by AgentCgroup [153, 72]. Each trace consists of the ordered sequence of tool calls issued during an agent task, including tool labels, command information where available, and tool call boundaries. We replay these traces under different isolation runtimes and collect execution time, resource usage, and host-level syscall activity for each tool call window. These measurements allow us to compare how the same logical agent task interacts with the underlying system across runtimes.

We measure across three isolation platforms: (1) native Linux containers (*runc*), (2) gVisor (*runsc-kvm*) release-20260520.0 using the KVM platform [82], and (3) Firecracker (*fc*) v1.15.1. These platforms represent increasingly strong isolation boundaries: *runc* shares the host kernel directly, *runsc-kvm* interposes at the system-call interface via a user-space kernel, and *fc* executes the workload inside a microVM. All experiments run on CloudLab [12] c220g5 nodes with 2.20GHz Intel Xeon Silver 4114 CPUs, 192GB memory, an Intel DC S3500 480GB SSD, a 1TB 7200RPM HDD, and a 10Gbps NIC. We run Ubuntu 22.04 with Linux kernel v6.1.

6.3.1 Tool-Call Replay Across Platforms

Our replay framework executes the extracted Haiku tool-call sequence inside each isolation platform. We measure only the execution time of the replayed tool-call sequence after the isolation environment has been initialized. Startup costs for creating the container, gVisor sandbox, or microVM are excluded. We run two sets of replay experiments.

First, we run each Haiku task in isolation under each platform and collect CPU and memory utilization metrics. These measurements motivate the runtime-selection problem by showing that the same logical agent task exhibits different resource behavior across

isolation platforms, as shown in Figure 6.1.

Second, we run profiling experiments across two modes under each platform: (a) solo and (b) corun. In the solo mode, the target task runs alone, and in the corun mode, the same target task runs concurrently with N co-located homogeneous neighboring workloads on the same platform. We use the ratio (corun/solo) between solo and corun execution time to compute slowdown at the task and tool call granularity. This slowdown is the observed interference risk signal used to validate whether higher EoR estimates correspond to higher performance degradation. For each replay, we also collect the syscall profile observed at the host layer during each tool call window.

Our two experiment sets use different resource configurations. For the resource-profiling runs (Figure 6.1), workloads are unpinned: *runc* and *runsc-kvm* containers may use all host cores with unbounded memory, and the *fc* microVM is configured with 20 vCPUs — matching the 20 physical cores — and 4 GB of memory.[1] For the solo and corun interference experiments, all platforms receive an identical allocation: the target workload is restricted to two cores (`cpuset` 0–1 for *runc* and *runsc-kvm*, a 2-vCPU microVM for *fc*) with 4 GB of memory, and each of the N neighboring workloads is pinned to cores 2–9 with 12 GB of memory.[2] We disable hyperthreading and partition the LLC between the target and neighbors using Intel CAT. Because the target and neighbors run on disjoint cores (with the single exception noted above), do not share SMT siblings, and occupy separate cache partitions, the corun slowdown we measure cannot arise from CPU time-sharing or LLC contention; it instead reflects contention on shared host-kernel state and other shared resources such as memory bandwidth and storage.

[1] We explicitly set the microVM machine configuration; when none is supplied, the VM launch path falls back to a default vCPU count, which would silently give the microVM a different CPU allocation than the containers.

[2] Two tasks use adjusted allocations, applied identically to solo and corun runs and across all runtimes: `pre-commit-2524`, whose `pip install` peaks near 10 GB and 40 cores, is given 4 cores (`cpuset` 0–3) and 12 GB; and `nv-ingest-71`, whose large microVM rootfs limits how many copies fit on the SSD, runs with N=3 neighbors. Because `pre-commit-2524`'s wider target cpuset overlaps the neighbor cores (2–9), the disjoint-core guarantee does not fully hold for that task, and its slowdown may include a CPU time-sharing component.

6.3.2 Syscall and Execution-Time Profiling

We profile each tool call using a common interface across platforms. The replay driver marks the start and end of every tool invocation and records its wall-clock execution time. In parallel, an eBPF profiler attaches to the kernel's `raw_syscalls:sys_enter` and `raw_syscalls:sys_exit` tracepoints to collect host-level syscall events. For each syscall, the profiler records the syscall identifier, process ID, return status, and latency. At the end of the run, syscall events are assigned to the tool call window in which they occurred. This produces per-tool-call syscall counts and latency statistics.

6.3.3 Estimating Shared-State Exposure

To estimate each task's shared-state exposure (§6.2) — which shared host-kernel locks a task's execution reaches, and how heavily, under each isolation runtime — we use host-level syscalls as a lightweight signal, avoiding expensive lock tracing for every full agent replay. This gives us a syscall-to-lock profile for the host kernel used in our experiments. The idea is that the runtime determines which host-level syscalls are issued during a tool call, while the syscall-to-lock profile estimates the shared host-kernel state exposure via the lock activity associated with those host-level syscalls. This is an approximation: the exact lock activity of a syscall can depend on arguments, object types, and execution context. However, it lets us estimate shared-state exposure without enabling expensive lock tracing [39] for every full agent replay.

6.3.4 EoR Calculation

Using the syscall-to-lock profile of §6.3.3, we now instantiate Equation 6.1 for agent tasks. For a task t running on platform p, let $S_{t,p}$ be the set of host-level system calls observed during the task's solo execution; we evaluate task-level isolation by aggregating system calls across all tool-call windows within the task, and compute:

$$\mathrm{EoR}(t,p) = \sum_{s \in S(t,p)} \lambda(t,p,s) \cdot R(s) \tag{6.2}$$

where $\lambda(t,p,s)$ is the interference magnitude of system call s, and $R(s)$ is the associated risk of one call of s.

We exclude blocking-by-design syscalls (*e.g.*, `epoll_pwait`, `poll`, `nanosleep`, `wait4`) from $S_{t,p}$, since their host-lock time is dominated by waiting rather than contention.

Interference Resource is the shared host-kernel state that can be contended across co-located tasks; we use shared kernel locks as its proxy.

Unit of Interference is the host-level system-call invocation. We denote the syscall identifier by s (*e.g.*, `read`, `write` or `mmap`). For each t and p, we count how many invocations of each syscall s are observed during solo execution.

Interference Magnitude is the solo issue rate of syscall s, computed by dividing the number of invocations of s by the execution time.

$$\lambda(t,p,s) = \frac{\texttt{count\_solo}(t,p,s)}{T_{\mathrm{solo}}(t,p)} \tag{6.3}$$

Risk is the shared-lock exposure per invocation of a host-level syscall s. We treat $R(s)$ as platform-independent: once a runtime issues syscall s to the host kernel, the syscall is serviced by the same host kernel and can reach the same shared kernel locks. Thus, different platforms affect EoR through the set and rate of host-level syscalls they generate, $\lambda(t,p,s)$, while the per-syscall risk $R(s)$ is obtained from a common syscall-to-lock profile. For each syscall s, we compute:

$$R(s) = \sum_{L} \frac{\texttt{hold\_ns}(s,L)}{\texttt{calls}(s,L) \cdot 10^9} \tag{6.4}$$

where L ranges over shared kernel locks reached by s, and $\texttt{calls}(s,L)$ is the number of invocations of s recorded in the same profiling run measured by $\texttt{hold\_ns}(s,L)$. Each ratio

is the average hold time on L per invocation of s; the 10^9 factor converts nanoseconds to seconds, so $R(s)$ has units of seconds per invocation.

Thus, $R(s)$ is not the risk of the syscall as a resource; it is the aggregate risk of the shared lock resources reached by one invocation of syscall s. This is an approximation: the exact locks and hold times of a syscall can depend on arguments, object types, and execution context. However, it captures the average shared-lock exposure of each host syscall and lets us compare runtimes without tracing locks during every full agent replay.

Multiplying the system-call issue rate in calls per second by the per-call risk in seconds per call gives a dimensionless EoR contribution. Thus, EoR estimates the fraction of execution exposed to shared host-kernel lock contention, using only solo syscall behavior and the syscall-to-lock calculation.

6.4 Results

In this section, we evaluate whether tool-call behavior can be used to estimate the isolation provided by different isolation platforms for agent tasks. We first measure the performance impact of tool-call co-location by comparing solo and co-running executions at both task (§6.4.1) and tool-call (§6.4.2) granularity, establishing the observed interference that an isolation metric should explain. We then evaluate whether task-level EoR (§6.4.3), computed from host-level syscall behavior and syscall-to-lock profiles, captures the shared host-kernel exposure associated with that interference.

6.4.1 Slowdown under co-location

Figure 6.2 shows task-level slowdown under co-location with $N = 10$ neighboring workloads. *runc* has the largest slowdown, *runsc-kvm* is intermediate, and *fc* has the smallest slowdown. The magnitude of the gap is substantial between runtimes: the median slowdown under *runc* is approximately 64%, compared with 25% for *runsc-kvm* and below 20%

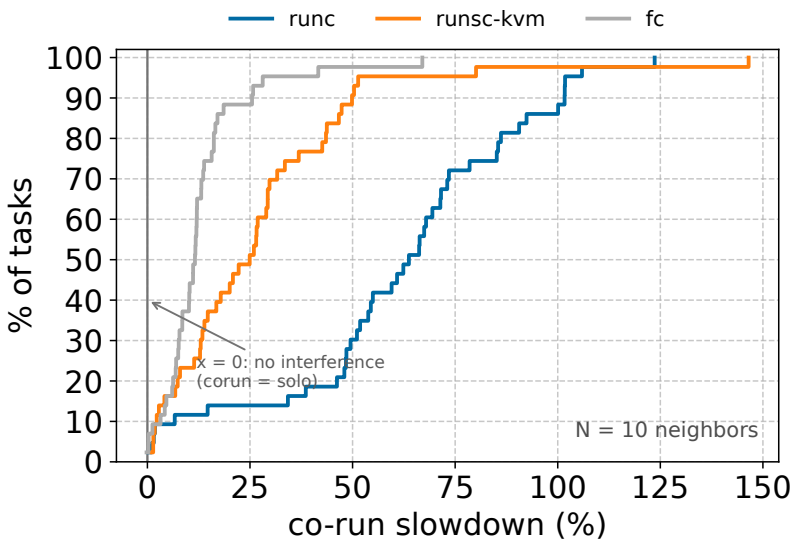


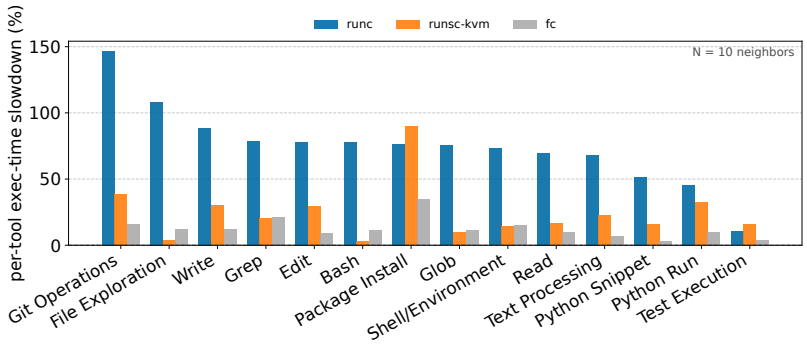
Figure 6.2: Task-level slowdown under co-location with N=10 neighboring workloads.

for *fc*. Thus, under the same co-location pressure, *runc* tasks slow down more than three times as much as *fc* tasks at the median.

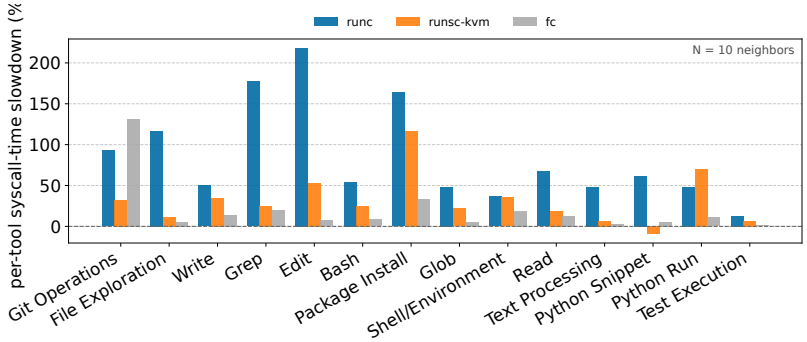
Crucially, some tasks experience little slowdown while others experience substantially larger degradation, particularly under *runc*. This task-level heterogeneity means that a fixed platform-level choice can over-provision low-risk tasks or under-protect high-risk ones. This implies the need for a per-task isolation signal that estimates the shared host state exposed by a task. Such a signal captures the isolation side of the placement trade-off; combined with performance targets and resource costs, it can guide selection of the *right* platform that keeps interference within an acceptable bound.

6.4.2 Tool-level Slowdown

We now break down the slowdown by tool category (Figure 6.3a), aggregating solo and corun execution time across all calls of each category, to understand which parts of the tool sequence are most affected. Under *runc*, several filesystem-facing categories experience large slowdowns, including Git operations, file exploration, writes, grep, edit, and Bash commands. Thus, the task-level slowdown is not spread uniformly across the tool sequence



(a) **Per-tool execution time slowdown.**



(b) **Per-tool syscall time slowdown.**

Figure 6.3: Tool-level slowdown under co-location with N=10 neighboring workloads.

but concentrated in specific tool categories.

runsc-kvm generally shows lower slowdown than *runc*, but not uniformly. Package installation, for example, slows down more under *runsc-kvm*, likely because it combines many filesystem, process, and package-cache operations that are mediated by the user-space kernel, amplifying runtime-specific overhead. *fc* shows the lowest slowdown for most tool categories, indicating lower end-to-end impact from co-located tool execution.

Next, we ask whether these end-to-end slowdowns are reflected in host-kernel syscall time, the component of execution that exercises shared kernel state. Figure 6.3b shows syscall-time slowdown for the same tool categories, computed by summing `count` $\times$ `avg_lat_ns` across all syscalls and tool calls in each category and comparing corun against solo.

For many tool categories, host syscall time increases under co-location, especially under *runc*. This shows that co-location affects a broad set of host syscall paths, not only the

Scope	n (data points)	Spearman ρ
Pooled (all platforms)	129	0.67
runc	43	0.46
runsc-kvm	43	-0.11 (n.s.)
fc	43	0.34

Table 6.1: Spearman rank correlation between EoR and measured corun slowdown (N=10), pooled across platforms and within each platform. *n.s.*: not significant ($p > 0.05$).

tools with the largest end-to-end slowdowns. However, syscall-time slowdown is tool-dependent and does not always match end-to-end slowdown. We minimize hardware-level interference by pinning CPUs, disabling hyperthreading, provisioning memory, and partitioning the LLC, but some shared residual channels may remain. We therefore treat host syscall behavior as one observable isolation dimension rather than a complete account of end-to-end slowdown.

Together, these results show that co-location slowdown is both tool-dependent and runtime-dependent. Execution-time slowdown shows where interference appears in the tool sequence, while syscall-time slowdown captures the shared host-kernel state dimension exposed through syscall paths. We next evaluate whether EoR can estimate this shared-state exposure from solo syscall behavior.

6.4.3 EoR

We evaluate the task-level EoR defined in §6.3.4 (Eq. 6.2). Since EoR is computed entirely from solo runs, the measured corun slowdown serves as a separate validation signal.

6.4.3.1 EoR–Performance Correlation

Our goal for this evaluation is not calibrated prediction of absolute slowdown; we evaluate EoR as a comparative isolation signal, testing for *rank prediction*: given two tasks, does the one with higher EoR have a higher likelihood of experiencing more slowdown under co-location? To test this, we use Spearman rank correlation [146], checking whether higher

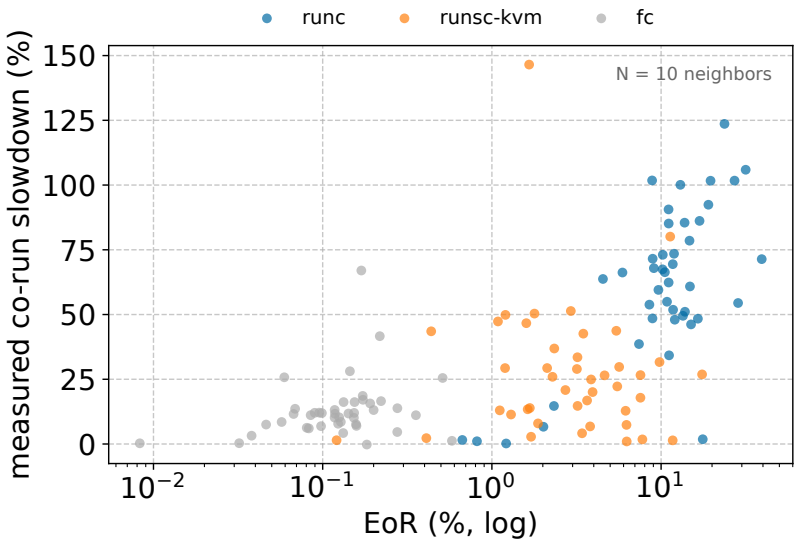


Figure 6.4: Task-level EoR versus measured corun slowdown.

task-level EoR corresponds to higher measured co-run slowdown. Table 6.1 shows the correlation pooled across all task–platform points and within each platform.

We observe that EoR is a strong rank predictor of slowdown when pooled across runtimes ($\rho = 0.67$, $n = 129$): tasks with higher EoR are substantially more likely to experience more slowdown, and this signal captures the large cross-platform differences in interference. Within *runc*—the platform that shares the host kernel directly—EoR remains predictive at the task level ($\rho = 0.46$), confirming that where host-lock sharing is the dominant channel, higher EoR reliably indicates higher relative slowdown. For *runsc-kvm* and *fc*, which limit syscalls to the host kernel, host-lock exposure is low by design; EoR's within-platform predictive power is correspondingly weaker (*fc*, $\rho = 0.34$) or statistically insignificant (*runsc-kvm*, $\rho = -0.11$). This pattern indicates that EoR predicts the shared host-lock component of interference specifically – dominant under *runc*, marginal under stronger isolation boundaries; the insignificant result for *runsc-kvm* indicates that there is no host-lock signal to detect, rather than an inverse relationship.

Platform	Median EoR (%)	Median slowdown (%)
runc	11.1	64
runsc-kvm	3.2	25
fc	0.13	12

Table 6.2: Median task-level EoR and measured corun slowdown by platform (N=10).

6.4.3.2 Cross-Platform EoR Ordering

Beyond per-task prediction, EoR reproduces the *cross-platform ordering* of interference from solo data alone. Table 6.2 reports median EoR and median measured slowdown per platform. EoR ranks the platforms in the same order as measured slowdown: *fc* < *runsc-kvm* < *runc*. This comparison is important because slowdown is the observed co-location outcome, while EoR is computed without running the workload under co-location.

The EoR gap is substantial: median EoR under *runc* is about 85× higher than under *fc* and about 3.5× higher than under *runsc-kvm*. This range is much wider than the corresponding slowdown range (64%, 25%, and 12%), indicating that EoR is not a linear predictor of absolute slowdown. Instead, it measures the host-shared-lock exposure created by each runtime. Thus, the same replayed agent tasks reach very different amounts of shared host-kernel state depending on the runtime.

With only three well-understood platforms, this ordering may seem unsurprising. Its value is that EoR derives the ranking from each platform's measured host-lock exposure rather than assuming it. The same procedure can therefore be applied when the ordering is not known a priori, such as with new runtimes, unfamiliar sandboxes, or different configurations of the same runtime. Here, the known *fc* < *runsc-kvm* < *runc* ordering serves as a sanity check that EoR captures the expected host-lock-exposure differences across these platforms.

Figure 6.4 shows that this separation holds at the task level, not only at the median: the platforms form ordered EoR bands that align with their slowdown bands, placing *runsc-kvm* between *runc* and *fc*. Because EoR is computed entirely from solo execution, it

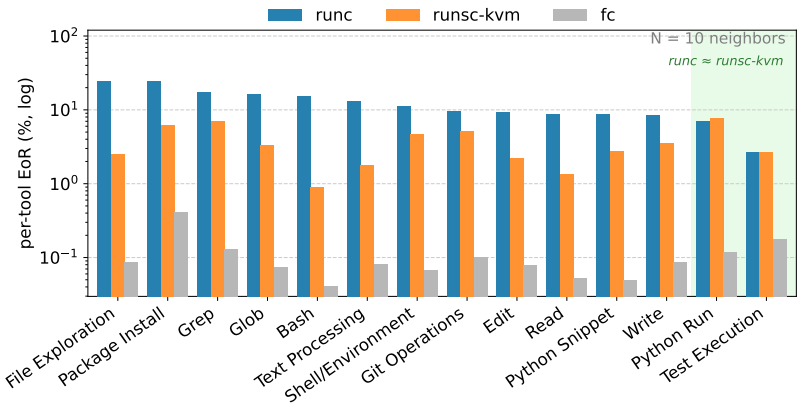


Figure 6.5: Tool-level EoR.

Scope	n (total tools)	Spearman ρ (exec)	Spearman ρ (syscall)
Pooled (all platforms)	42	0.82 ($p<0.001$)	0.74 ($p<0.001$)
runc	14	0.52 ($p=0.059$)	0.41 ($p=0.144$)
runsc-kvm	14	0.53 ($p=0.049$)	0.52 ($p=0.059$)
fc	14	0.36 ($p=0.210$)	0.23 ($p=0.422$)

Table 6.3: Spearman rank correlation between per-tool EoR and measured per-tool corun slowdown (N=10), using execution-time (Figure 6.3a) and syscall-time (Figure 6.3b) slowdown. Each within-platform correlation uses 14 tool categories.

can provide the isolation side of a placement decision by ranking platforms according to shared-host state exposure, without requiring direct corun measurements for every task.

6.4.3.3 Tool-Level EoR

Task-level EoR aggregates over an entire agent trajectory. We now compute EoR at tool-call granularity, aggregating solo syscall profiles within each of the 14 tool categories, and ask whether EoR ranks *tools* by interference exposure the way it ranks tasks and platforms.

Figure 6.5 shows per-tool EoR across platforms. We make two key observation. First, the isolation benefit is tool-dependent. For file and I/O-heavy tools such as File Exploration and Read, *runsc-kvm* has substantially lower EoR than *runc*, by about 5-10×. In comparison, for Test Execution and Python Run, EoR under *runsc-kvm* is comparable to *runc*, indicating that *runsc-kvm* does not substantially reduce host-lock exposure for these tool categories.

Second, the relative EoR of tools changes across platforms. File Exploration has one of

the highest EoRs under *runc*, but drops substantially under *runsc-kvm*. In contrast, Python Run has relatively low EoR under *runc*, but is among the highest under *runsc-kvm*. Thus, the isolation benefit of a platform depends on the tools an agent executes. Thus, the same isolation platform may provide different benefits depending on a task's tool mix.

Table 6.3 reports the rank correlation between per-tool EoR and the tool-level slowdowns of §6.4.2. Pooled across platforms, the correlation is strong (ρ=0.82 against execution-time slowdown, ρ=0.74 against syscall-time slowdown). As with the task-level result, the pooled correlation partly reflects differences across platforms. A harder test is whether EoR can rank tools within each platform.

Unlike the task-level result, where *runsc-kvm* showed no within-platform signal (ρ=−0.11), the tool-level correlations are positive on all three platforms. The execution-time correlations are similar for *runc* (ρ=0.52) and *runsc-kvm* (ρ=0.53). This suggests that aggregation by tool category reduces some per-task variation and reveals a rank signal that is less visible at task granularity. However, with only n=14 tool categories per platform, statistical power is limited, so we interpret these within-platform results as directional rather than conclusive.

The correlation is weakest under *fc* (ρ=0.36 for execution time). As Figure 6.5 shows, *fc* keeps host-lock exposure low across all tool categories. Within this low-exposure regime, differences in end-to-end slowdown may increasingly reflect interference sources outside the shared host-lock state captured by the current EoR instantiation.

6.5 Discussion and Future Work

EoR complements resource partitioning. CPU pinning, cgroups, and cache or bandwidth partitioning control resource allocation, but they do not eliminate interference through shared kernel state. Co-located workloads may still contend on allocator structures, or filesystem-journaling paths even when CPU, memory, and cache resources are controlled.

EoR captures this residual shared kernel-state interference and can serve as the isolation side of a future placement policy, while allocation controls manage partitionable resources within the chosen isolation boundary.

From prediction to placement. EoR provides the isolation side of the placement trade-off, but runtime selection must balance isolation, performance, and efficiency. A simple comparative policy is to select the platform with the lowest EoR for the task. If multiple platforms have comparable EoR, the policy selects the most resource-efficient one. Under this rule, a platform provides *enough* isolation when no alternative offers a meaningfully lower EoR; otherwise, the lower-EoR platform shows that stronger isolation is available. A future policy would refine this rule by combining EoR with slowdown targets and runtime overheads to choose a runtime that provides sufficient isolation without over-isolating low-risk tasks or under-isolating high-risk ones. This requires calibrating EoR against performance targets and extending it beyond the current shared-kernel-state signal to other interference dimensions.

Scaling and tool non-determinism. Applying EoR at scale raises two practical questions. First, EoR requires profiling tool behavior, which appears costly for a large tool ecosystem. However, the expensive component — the syscall-to-lock profile $R(s)$ — is a property of the host kernel, measured once and reused across all tools and workloads; per-task EoR then needs only cheap solo syscall counts (λ), with no co-run or lock tracing. Because an agent's toolset is bounded and tools recur across tasks, these per-tool profiles amortize. Second, a bounded tool set does not bound tool *behavior*: a single tool such as a shell or test runner can issue very different syscalls depending on its arguments and inputs, which are not known in advance. EoR is defined as an *expectation* precisely to accommodate this: rather than a point prediction, a tool can be characterized by a distribution of syscall behavior collected over many invocations, and placement can reason about expected interference.

— 7 —

Related Work

> If we knew what it was we were doing, it would not be called research, would it?
>
> *Albert Einstein*

In this chapter, we discuss prior work in the context of this dissertation's contributions. We organize the discussion by theme, following the arc of the dissertation itself: the design of lightweight isolation platforms (§7.1); measurement of kernel footprint and virtualization performance, related to Chapter 3 (§7.2); analysis of kernel locks and shared kernel state, related to Chapter 4 (§7.3); measurement and prediction of cross-workload interference (§7.4); metrics for isolation and security, related to Chapter 5 (§7.5); and isolation for agentic workloads, related to Chapter 6 (§7.6).

7.1 Lightweight Virtualization and Isolation Platforms

MicroVMs and secure containers. This dissertation studies Firecracker [16] and gVisor [18], both KVM-based sandboxes attempting to compete on performance with traditional LXC containers [31], on which Docker is built. New sandboxing systems with similar goals include Kata Containers [21] (which attempt to provide "the speed of containers, the security of VMs"), Nabla containers [51], and LightVM [105]. Manco *et al.* [105] argue that traditional containers are insecure because malicious code has access to a large

attack surface (all Linux system calls); many of the new systems are structured much like Drawbridge [117], in an attempt to narrow this interface. Narrowed interfaces are then typically protected with stricter seccomp [24] filters. This design philosophy is reflected in many cloud offerings such as AWS Nitro [9] and Cloud Hypervisor [11], which minimize device emulation and reduce interactions with the host.

Unikernels and library OSes. A more aggressive point in the design space minimizes the workload's dependence on the shared host kernel. Unikernels [103, 97] compile an application together with only the OS code paths it needs into a minimal, single-purpose kernel image, and library OS designs such as Drawbridge [117], Graphene [134], and X-Containers [128] move OS services into the workload's own address space, reducing reliance on shared host-kernel state. These designs trade compatibility and operational maturity for smaller shared footprints; our measurements quantify precisely the shared footprint that such designs aim to eliminate, providing an empirical baseline for the isolation they buy.

Runtime and language-based isolation. Serverless and edge platforms increasingly isolate tenants within a single process using language runtimes, such as V8 isolates [136] and WebAssembly-based sandboxes [143, 129]. These mechanisms offer very low startup cost, but all tenants share not only the host kernel but the runtime process itself, placing them at the far-sharing end of the isolation spectrum this dissertation studies. Our framework applies naturally to such platforms: their interference points simply lie in the shared runtime as well as the shared kernel.

Hardware-supported isolation. At the other end of the spectrum, confidential-computing extensions such as AMD SEV-SNP [126], Intel TDX [133], and Arm CCA [41] protect a workload's memory and execution state, even from the host itself. These mechanisms root confidentiality and integrity in hardware rather than in the host kernel: a workload's guarantees hold even if the host kernel is fully compromised, so correctness no longer depends on trusting the kernel objects the workload shares with it. The workload still shares

the underlying hardware, and the host kernel still manages it through many of the same kernel objects our measurements target, such as scheduling structures, memory allocation for encrypted pages, and I/O paths, so interference through shared host resources is not eliminated, only separated from the confidentiality guarantee. This is the dimension our measurements and the metric capture.

Applicability. Our measurement methodologies are not specific to gVisor and Firecracker; the coverage analysis of Chapter 3 and the lock-based object analysis of Chapter 4 could be applied to these other isolation platforms as well.

7.2 Measuring Kernel Footprint and Virtualization Performance

Both gVisor and Firecracker restrict the set of system calls available to improve security and isolation. Narrowing the interface of a platform will not necessarily improve security, unless it reduces the footprint of host code likely to contain vulnerabilities, which is why Chapter 3 focuses on understanding the kernel footprint underneath each virtualization system. Bottomley [51] takes a similar view and measures the footprint of various sandboxes at function granularity; we explore footprint at line and branch granularity, using lcov [23]. gVisor implements the network stack in the Sentry; while the work of Williams *et al.* [63] shows that moving subsystems out of the host kernel can significantly reduce kernel footprint, our results show gVisor and LXC have a surprisingly large coverage overlap in `/net`.

In addition to measuring code footprint, Chapter 3 also evaluates performance, building on prior virtualization performance analysis studies [38, 95, 141, 73, 142, 150]. Our work differs from past studies in that it focuses on the performance impact of architectural choices and identifies the kernel code differences that play a role in performance.

7.3 Kernel Locks and Shared Kernel State

Kernel locks. Multiple tools have been proposed to detect race conditions and deadlocks. Eraser [124] proposes a novel lockset algorithm for dynamically detecting data races in multithreaded programs. LockDoc [102] performs dynamic trace-based kernel analysis to infer locking rules for data structures and identify locking violations. Unlike these tools, which focus on correctness and bug detection, Chapter 4 uses locking behavior as a proxy to quantify cross-workload object interference and architectural isolation.

Scalability and commutativity. Min *et al.* [107] identified kernel objects that limit filesystem performance and scalability. Clements *et al.* [55] proposed the scalable commutativity rule: interface operations that are order-independent can be implemented to allow scalability. Multikernel [45] proposes a new architecture for scaling multicore systems by avoiding any inter-core sharing. While these works focus on scalability, we examine how modern isolation mechanisms expose shared kernel objects across diverse workloads.

7.4 Measuring and Predicting Interference

Interference measurement. KIT [100] detects inter-container functional interference by comparing the system call traces of a container across two different executions (running with and without another container). Other works have studied and measured performance interference for workloads by either characterizing workloads [69, 53] under different stress levels or by proposing an analytical model [140]. In contrast, Chapter 4 analyzes kernel object sharing directly, exposing structural interference rooted in shared kernel state, rather than observing its symptoms.

Interference prediction and scheduling. A complementary line of work predicts interference to drive placement. Paragon and Quasar [66, 67] classify workloads online against a reference set to predict interference and guide QoS-aware scheduling at datacenter

scale, characterizing interference largely through hardware-level contention signals—cache capacity, memory bandwidth—per resource class; providers additionally use historical workload data to inform placement [62]. In contrast, EoR characterizes interference through *software*-level shared kernel state–the objects and locks a workload exercises in the host kernel–providing a workload and platform-specific isolation value that such schedulers could use as an additional placement signal alongside hardware-contention measures (Chapter 5).

7.5 Isolation and Security Metrics

Attack surface. The closest metric in spirit to EoR is the attack surface metric [104], which quantifies how exposed a system is to potential attacks–how *attackable* it is–by counting attack vectors such as entry and exit points (*e.g.*, system calls) and channels (*e.g.*, open ports). The two metrics are complementary: the attack surface captures a system's static exposure to an adversary, while EoR captures the impact of interference in the presence of co-running workloads through shared resources (Chapter 5). Both can be used together to compare the isolation guarantees of platforms. Notably, much of the lightweight virtualization work in §7.1 is motivated by attack surface reduction; Chapter 3 shows that a narrowed interface does not necessarily imply a smaller host-kernel footprint, suggesting that interface-level exposure metrics alone are incomplete.

Formal frameworks for sharing and isolation. OSmosis [37] provides a formal framework for reasoning about which resources isolation mechanisms share or isolate across a taxonomy of isolation abstractions. It answers *what is shared* by construction of the abstraction; EoR complements it by measuring *how much a given workload exercises* that shared surface and the risk this poses under co-location.

7.6 Isolation for Agentic Workloads

AI agents execute model-directed actions through tools, and recent systems work has begun to address the safety and isolation of this execution layer. Sandboxing systems for LLM-generated and externally supplied code isolate code interpreters and tool runtimes in containers or microVMs [36, 81], and multi-tenant tool servers such as MCP [40] centralize tool execution, raising isolation questions across agent sessions [83, 119]. Prior work on agent security focuses primarily on containment, preventing prompt-injected or malicious agents' actions from escaping the sandbox [148, 64], whereas Chapter 6 addresses a complementary question: among safe placements, which isolation platform provides *enough* isolation for a given agent task? Work on serverless and sandbox cold-start optimization [112, 71] is also relevant, as agents' bursty, short-lived tool invocations inherit these startup costs; our results show that initialization additionally intensifies object-level sharing (Chapter 4). Closer to our per-tool resource profiles, characterizations of serverless workloads at scale [127] use observed resource-usage patterns to drive provisioning policy, though at function rather than tool granularity.

— 8 —

Conclusions and Future Work

> One never notices what has been done; one can only see what remains to be done.
>
> *Marie Curie*

This dissertation revisited a question usually answered by conventional wisdom rather than measurement: *how much isolation does a platform actually provide to a given workload?* Across all studies outlined, we showed that isolation is a workload-dependent and quantifiable property, determined by how a workload exercises shared host-kernel state and by the neighbors it runs alongside. We conclude by summarizing the contributions of each chapter (§8.1), reflecting on the broader lessons of this work (§8.2), and outlining directions for future research (§8.3).

8.1 Summary

How do platforms use the host kernel? Chapter 3 examined how isolation platforms use the host kernel. These platforms form a spectrum as they move functionality out of the host kernel and into an isolated guest environment – gVisor handles many system calls in a user-mode Sentry process, while Firecracker runs a full guest operating system in each microVM – and we asked how much their use of host-kernel functionality actually

varies, using fine-grained code coverage and microbenchmarks targeting different kernel subsystems. Despite moving much functionality out of the kernel, both Firecracker and gVisor execute substantially more kernel code than native Linux, and gVisor and Linux containers execute substantially the same code, although with different frequencies. These findings established the empirical foundation of this dissertation: the boundary between a workload and the shared kernel is shaped by both platform design and workload behavior, and cannot be assessed from platform architecture alone.

What kernel state is shared? Chapter 4 moved from how much kernel code is executed to what kernel state is shared. Even platforms that move functionality away from the host kernel still rely on kernel objects to implement core system functionality – memory management, file systems, I/O, and scheduling – so isolation is not determined solely by whether a platform virtualizes a resource, but by which kernel objects remain shared and how workloads interact with them. Leveraging the observation that access to shared kernel objects is synchronized, we built LIMA, a fine-grained tracing tool that combines dynamic tracing of kernel locking events with static analysis mapping locks to the data structures they protect, and applied it to a comparative analysis of LXC, gVisor, and Firecracker across a broad set of workloads. Our analysis found that the file-system journal and the kernel page allocator are the most common sources of shared data, and revealed distinct object-sharing profiles for each platform: isolation is a spectrum shaped by platform design choices that redistribute and reshape object-level sharing.

Quantifying isolation. Chapter 5 formalized these observations into a metric. Without standard methods to measure isolation and interference, we cannot determine how much isolation a workload gets in a given environment, making it impossible to quantitatively compare platforms or environments. We defined a formal system model of a cloud environment and leveraged it to design Expectation of Risk (EoR), a metric that quantifies the isolation a workload receives by combining interference magnitude with risk at each shared interference point. Instantiated with shared kernel locks, EoR correlates with mea-

sured performance change under interference, demonstrating that it can enable meaningful comparison across isolation platforms – and that isolation can be treated as a measurable quantity rather than a qualitative judgment.

Isolation for agentic workloads. Chapter 6 applied EoR to an emerging workload class that is changing the isolation landscape: AI agents. Unlike traditional workloads, agents execute heterogeneous and dynamic sequences of tools that read and write files, spawn processes, install packages, and interact with external services, so the question is no longer simply whether to isolate agent execution, but how to choose an isolation boundary that balances isolation, performance, and efficiency for a dynamic tool set. We characterized agent tasks under co-location, finding substantial and heterogeneous slowdown across platforms, and instantiated EoR for shared host-kernel exposure using host-level syscall profiles and shared kernel locks, evaluated on replayed agent tool-call traces. EoR ranks runtimes by their exposure to shared host-kernel state, and this ranking aligns with measured co-location slowdown, especially when shared host-kernel state is the main source of interference – providing a first step toward tool-aware selection of isolation platforms for agentic workloads.

One argument at increasing depth. Together, these chapters trace a single argument: from measuring how much of the shared kernel each platform exercises (Chapter 3), to identifying which kernel objects workloads share (Chapter 4), to quantifying that shared exposure (Chapter 5), and finally applying the quantity to a new and evolving class of workloads (Chapter 6).

8.2 Lessons Learned

Predicting isolation is hard, and security isolation is hardest. A recurring experience across this work is that precisely predicting isolation is very hard. Interference depends on dynamic factors – which neighbors are present, what they are doing, and when – so

even a well-grounded signal like EoR is most reliable as a comparative measure: ordering environments and platforms rather than predicting the magnitude of slowdown for a single deployment. Our evaluations reflect this: EoR tracks isolation differences across platforms more strongly than variation among tasks within a single platform, where residual factors beyond shared-kernel exposure dominate. The security dimension of isolation is harder still. Performance interference is continuous and observable – we can measure slowdown – but security interference is rare, adversarial, and often invisible until exploited, which makes assigning risk values to security interference points fundamentally more speculative. Moreover, security failures are often less graceful than performance failures: a single inadequately protected shared-state object can enable substantial data disclosure or compromise an entire virtual machine, whereas performance interference manifests as degradation – increased latency or reduced throughput – well before complete service failure.

For security, we can take a conservative stance: when we cannot measure a risk, it is reasonable to assume the worst and pay for stronger isolation than strictly necessary. But we do not have to be conservative everywhere. If we could label kernel state as sensitive or not–the same classification that state-minimizing designs such as address-space isolation must make when choosing which state to unmap–we could reserve high risk values for interference points that touch sensitive data and assign lower values everywhere else. Classification alone, however, is insufficient: determining the appropriate values for each class remains an open calibration problem.

Object-level analyses like ours are a plausible starting point for such tagging, but they answer only part of the question: our analysis identifies which kernel objects are shared, not what the consequences are if one leaks. Assessing that impact requires complementary techniques–such as information–flow or taint analysis–that can trace what data a shared object exposes and to what degree. Moreover, an attacker need not respect object boundaries–memory-safety violations such as buffer overruns can reach objects outside a

workload's intended set–so security reasoning must ultimately extend to the address-space level. We believe the underlying asymmetry is intrinsic: performance isolation can be measured and optimized, while security isolation must be modeled conservatively until sensitivity can be identified and exploitability estimated.

Shared kernel state deserves more attention than it receives. Discussions of multi-tenant isolation are dominated by hardware mechanisms: cache partitioning, side-channel defenses, confidential computing, while the kernel's shared-state surface, though heavily engineered, is rarely measured. Yet our measurements consistently point to a quieter and larger shared surface: the Linux kernel itself. The kernel is a single shared entity of tens of millions of lines, and every platform we studied, however minimal its design, ultimately routes workloads into shared kernel code paths and shared kernel objects. Two containers with disjoint resource limits still meet in the inode cache; a microVM that narrows its device model still contends on the host's memory allocator and journal. Isolation platforms should treat kernel-state sharing as a first-class design concern — without that, hardware protections address only part of the problem.

Toward isolation-aware development. Isolation platforms evolve constantly; interfaces are narrowed, functionality is moved, device models are redesigned – and these changes are made in the name of isolation. Yet unlike performance, where every change can be checked against benchmarks, there is no metric to tell designers whether a change actually improved isolation, by how much, or at what cost. Isolation is argued about at design time and validated after the fact, but never measured along the way. The methods in this dissertation can fill this gap: coverage analysis (Chapter 3) can show how a change affects the host-kernel footprint; LIMA (Chapter 4) can flag when a new feature adds contention on hot kernel objects; and EoR (Chapter 5) can catch isolation regressions the way performance regressions are caught today. New designs that aim to reduce shared kernel state; address-space isolation, per-tenant copies of hot kernel data, per-tenant kernels – need exactly this visibility to decide which state to privatize first. We hope this work

provides both the motivation and the tools to make isolation a measured part of the design process.

Personal Reflections. Some lessons were more about the process of conducting research. Tracing fine-grained kernel behavior is painful: lock-level instrumentation is noisy, expensive to collect, and easy to misread, and much of the effort in this dissertation went into making that signal trustworthy before anything could be built on it. Not every tooling choice felt certain at the outset: clangd was well suited for the task, but no one had used it this way before, which made starting with it feel uncertain; it proved remarkably fruitful, and others exploring kernel code may find it similarly useful. Formulating the metric precisely was itself clarifying. I started by asking what the root cause of interference is, and only then proceeded to quantification–a sequence that shaped EoR more than any later refinement did. It is also easy to become fixated on a single line of thought; another pair of eyes, and a willingness to actually listen, can help immensely–some of the turning points in this work came from feedback rather than from further effort along the same path. Systems research is mostly slow and rewards patience; results that look simple in a chapter often represent months of infrastructure and dead ends. Finally, writing was not a record of finished thinking but part of the thinking itself. Many ideas here only became clear when they had to be written down.

8.3 Future Work

Beyond locks: lockless sharing and hardware sharing. LIMA observes sharing through kernel synchronization, which misses coordination that does not take locks: read-copy-update (RCU), atomic operations, *etc.* These mechanisms are designed to reduce contention, but they do not eliminate data sharing (*e.g.*, RCU readers still traverse shared structures). Extending the object analysis to lockless mechanisms, for example by tracing RCU critical sections or sampling cache-coherence traffic, would complete the picture of kernel-state

sharing and test whether our lock-derived findings, such as the dominance of allocator and journaling objects, hold under a wider lens.

Sharing also extends below software: co-located workloads contend on hardware resources such as shared caches, memory bandwidth, and TLBs that no kernel object analysis can see. Incorporating hardware contention signals into EoR alongside its software interference points would let a single value account for both sources of exposure, and would test whether platforms that rank well on shared kernel state also rank well when hardware contention is included.

Instantiating the security dimensions of EoR. Our evaluations instantiate EoR for performance interference; the framework's security dimensions (Chapter 5) remain outlined rather than computed. Channel-specific leakage measures from the side-channel and quantitative information-flow literature are natural candidates for the magnitude and risk values of security interference points. Composing such measures within EoR would provide a common framework for comparing performance and security isolation and would directly confront the modeling challenges identified above.

EoR-driven deployment: from comparison to decision. EoR currently compares isolation platforms; the next step is to let it drive deployment decisions. Quantification is what makes *how much isolation is enough* answerable: once isolation is a measured value, enough becomes a threshold; the weakest platform whose exposure for a given task stays within what that workload's sensitivity and neighborhood can tolerate rather than a fixed worst-case choice. These decisions have two parts: where a workload runs, and how it is isolated. For the *where*, an interference-aware scheduler could use task-level EoR as a placement signal, co-locating low-exposure workloads densely, reserving stronger platforms for high-exposure tasks, and re-evaluating placements as neighborhoods change. Achieving this requires a cheap, online estimation of EoR (our current pipeline is offline), incremental updates as workloads arrive and depart, and integration with the resource signals schedulers already use.

Placement systems for traditional workloads predict interference from execution history, which requires the workload to have run before; an agent's toolset, in contrast, is declared before first execution, and because we profile resource usage per tool rather than per task, the profiles of familiar tools can be combined to estimate the exposure of a task that has never run. For the *how*, agents sharpen the question of choosing the isolation boundary itself: a runtime could select or compose isolation environments per task or per tool at invocation time, balancing EoR against startup latency and resource efficiency, and adapt as an agent's toolset evolves. A complete system would make the two decisions jointly, since the best placement depends on the platform's exposure and vice versa. As agents become persistent tenants of shared infrastructure, we expect isolation selection to shift from a static deployment decision to a runtime one; this dissertation provides the measurement foundation that such systems will need.

8.4 Closing Remarks

The mechanisms of isolation are quantitative – resource limits, partition sizes, narrowed interfaces – but the property itself has been discussed in the language of promises: platforms promise isolation, tenants trust it, and conventional wisdom ranks it, without a way to measure how much of it a given workload actually receives. This dissertation has argued that isolation can be treated like the mechanisms beneath it: as a property one can measure, compare, and design for. We hope the tools, metric, and evidence assembled here help move the field in that direction, so that the next generation of isolation platforms is built not on what we assume is shared, but on what we know.

Bibliography

[1] Amazon ec2. https://aws.amazon.com/ec2/, 2006.

[2] Amazon s3. https://aws.amazon.com/s3/, 2006.

[3] Amazon virtual private cloud. https://aws.amazon.com/vpc/, 2009.

[4] Amazon ec2 dedicated hosts. https://aws.amazon.com/ec2/dedicated-hosts/, 2011.

[5] Amazon elastic container service. https://aws.amazon.com/ecs/, 2014.

[6] Aws lambda. https://aws.amazon.com/lambda/, 2014.

[7] Aws confidential computing. https://aws.amazon.com/confidential-computing/, 2017.

[8] Aws fargate. https://aws.amazon.com/fargate/, 2017.

[9] Aws nitro system. https://aws.amazon.com/ec2/nitro/, 2017.

[10] Chapter 1. introduction to linux containers. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers, 2019.

[11] Cloud hypervisor. https://www.cloudhypervisor.org/, 2019. Accessed July 2026.

[12] Cloudlab. https://www.cloudlab.us/, 2019.

[13] Cloudsuite. https://github.com/parsa-epfl/cloudsuite, 2019.

[14] Docker. https://docs.docker.com/, 2019.

[15] Docker seccomp. https://docs.docker.com/engine/security/seccomp/, 2019.

[16] Firecracker documentation. https://github.com/firecracker-microvm/firecracker/tree/master/docs, 2019.

[17] Firecracker seccomp. https://github.com/firecracker-microvm/firecracker/blob/v0.19.1/vmm/src/default_syscalls/filters.rs, 2019.

[18] gvisor documentation. https://gvisor.dev/docs/, 2019.

[19] gvisor mm. https://github.com/google/gvisor/tree/master/pkg/sentry/mm, 2019.

[20] iperf3. https://github.com/esnet/iperf, 2019.

[21] Kata containers. https://katacontainers.io/, 2019.

[22] Kvm. https://wiki.archlinux.org/index.php/KVM, 2019.

[23] lcov. http://ltp.sourceforge.net/coverage/lcov.php, 2019.

[24] Linux seccomp. http://man7.org/linux/man-pages/man2/seccomp.2.html, 2019.

[25] Nabla containers. https://nabla-containers.github.io/, 2019.

[26] Qemu. https://wiki.archlinux.org/index.php/QEMU, 2019.

[27] sysbench. https://github.com/akopytov/sysbench, 2019.

[28] Docker frequently asked questions (faq). https://docs.docker.com/engine/faq/, 2020.

[29] firecracker-containerd. https://github.com/firecracker-microvm/firecracker-containerd, 2020.

[30] gvisor performance guide. https://gvisor.dev/docs/architecture_guide/performance/, 2020.

[31] Linux containers. https://linuxcontainers.org/lxc/introduction/, 2020.

[32] Lxc. https://en.wikipedia.org/wiki/LXC, 2020.

[33] Open container initiative. https://www.opencontainers.org/, 2020.

[34] Language server protocol. https://microsoft.github.io/language-server-protocol/, 2024.

[35] Distributive property. https://en.wikipedia.org/wiki/Distributive_property, 2025.

[36] 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.

[37] Sidhartha Agrawal, Shaurya Patel, Arya Stevinson, Linh Pham, Ilias Karimalis, Hugo Lefeuvre, Aastha Mehta, Reto Achermann, and Margo I. Seltzer. Comparing isolation mechanisms with osmosis. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems*, PLOS '25, page 1–9, 2025.

[38] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance Evaluation of Microservices Architectures Using

Containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, Boston, MA, 2015.

[39] Anjali and Michael M. Swift. Locked in, leaked out: Measuring isolation via kernel locks. https://arxiv.org/abs/2507.21248, 2025.

[40] Anthropic. Introducing the model context protocol. `https://www.anthropic.com/news/model-context-protocol`, 2024.

[41] Arm confidential compute architecture. `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`, 2021.

[42] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 375–388, 2011.

[43] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. https://arxiv.org/abs/2505.20411, 2025.

[44] 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, 2003.

[45] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of*

the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, page 29–44, 2009.

[46] Antoine Beaupré. Updates in container isolation. `https://lwn.net/Articles/754433/`, May 2018.

[47] Emir Beganović. Your container is not a sandbox. `https://emirb.github.io/blog/microvm-2026/`, 2026.

[48] Teofil Bodea, Masanori Misono, Julian Pritzi, Patrick Sabanic, Thore Sommer, Harshavardhan Unnibhavi, David Schall, Nuno Santos, Dimitrios Stavrakakis, and Pramod Bhatotia. Trusted ai agents in the cloud. https://arxiv.org/abs/2512.05951, 2025.

[49] Paolo Bonzini. An introduction to lockless algorithms. `https://lwn.net/Articles/844224/`, 2023.

[50] Gianluca Borello. Container isolation gone wrong. `https://sysdig.com/blog/container-isolation-gone-wrong/`, 2017.

[51] James Bottomley. Measuring the horizontal attack profile of nabla containers. `https://blog.hansenpartnership.com/measuring-the-horizontal-attack-profile-of-nabla-containers/`, July 2018.

[52] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, nov 1997.

[53] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, 2019.

[54] clangd language server. https://github.com/clangd/clangd, 2024.

[55] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 1–17, 2013.

[56] Google Cloud. Choose your agentic ai architecture components. https://docs.cloud.google.com/architecture/choose-agentic-ai-architecture-components, 2026.

[57] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), page 185–196, 1965.

[58] Jonathan Corbet. A call to reconsider address-space isolation. https://lwn.net/Articles/909469/, 2022.

[59] Jonathan Corbet. Generalized address-space isolation. https://lwn.net/Articles/886494/, 2022.

[60] Jonathan Corbet. Lockless algorithms for mere mortals. https://lwn.net/Articles/827180/, 2024.

[61] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, 2017.

[62] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, 2017.

[63] Dan Williams and Ricardo Koller and Brandon Lum. Say Goodbye to Virtualization for a Safer Cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018.

[64] Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: a dynamic environment to evaluate prompt injection attacks and defenses for llm agents. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, 2024.

[65] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 23–33, 2013.

[66] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013.

[67] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[68] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, page 473–488, 2016.

[69] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 599–613, 2017.

[70] Azhar Desai. The firecracker virtual machine monitor. `https://lwn.net/Articles/775736/`, January 2019.

[71] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, 2020.

[72] eunomia lab. Agentcgroup: Understanding and controlling os resources of ai agents. https://github.com/eunomia-bpf/agentcgroup, 2026.

[73] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), 2015.

[74] Dan Fernández. What is an ai agent? from chatbots to autonomous systems. https://edera.dev/stories/what-is-an-ai-agent-from-chatbots-to-autonomous-systems, 2025.

[75] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, 2019.

[76] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[77] G. Scott Graham and Peter J. Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), page 417–429, 1971.

[78] Brendan Gregg. Bpf performance tools: Linux system and application observability (chapter 2). Addison-Wesley Professional, 2019.

[79] Brendan Gregg. Flame graphs. `http://www.brendangregg.com/flamegraphs.html`, 2019.

[80] Brendan Gregg. The return of the frame pointers. `https://www.brendangregg.com/blog/2024-03-17/the-return-of-the-frame-pointers.html`, 2024.

[81] gvisor. `https://gvisor.dev/`, 2024.

[82] Platform guide. `https://gvisor.dev/docs/architecture_guide/platforms/`, 2026.

[83] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Trans. Softw. Eng. Methodol.*, February 2026. Just Accepted.

[84] Hang Huang, Honglei Wang, Jia Rao, Song Wu, Hao Fan, Chen Yu, Hai Jin, Kun Suo, and Lisong Pan. vkernel: Enhancing container isolation via private code and data. *IEEE Transactions on Computers*, 73(7):1711–1723, 2024.

[85] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, July 2018.

[86] iproute2 project. tc(8) — linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[87] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. https://arxiv.org/abs/1902.03383, 2019.

[88] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 729–742, 2014.

[89] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[90] Michael Kerrisk. cgroups(7) — linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[91] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[92] Juhee Kim, Xiaoyuan Liu, Zhun Wang, Shi Qiu, Bo Li, Wenbo Guo, and Dawn Song. The attack and defense landscape of agentic ai: A comprehensive survey. https://arxiv.org/abs/2603.11088, 2026.

[93] Colin Ian King. stress-ng - a tool to load and stress a computer system. https://manpages.ubuntu.com/manpages/jammy/man1/stress-ng.1.html, 2025.

[94] Chad Kittel, Clayton Siemens, Hemavathy Alaganandam, James Lee, Ritesh Modi, Mahdi Setayesh, Mark Taylor, and Yaniv Vaknin. Ai agent orchestration patterns, 2026.

[95] Zhanibek Kozhirbayev and Richard O. Sinnott. A Performance Comparison of Container-based Technologies for the Cloud. *Future Generation Computer Systems*, 68, 2017.

[96] M. Krohn and E. Tromer. Noninterference for a practical difc-based operating system. In *2009 30th IEEE Symposium on Security and Privacy*, pages 61–76, 2009.

[97] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (*EuroSys*), 2021.

[98] Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, page 341–352, 2008.

[99] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 22*), pages 465–484, July 2022.

[100] Congyu Liu, Sishuai Gong, and Pedro Fonseca. Kit: Testing os-level virtualization for functional interference bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 427–441, 2023.

[101] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[102] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

[103] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[104] Pratyusa K. Manadhata and Jeannette M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[105] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, 2017.

[106] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. Rcu usage in the linux kernel: Eighteen years later. *SIGOPS Oper. Syst. Rev.*, 54(1):47–63, August 2020.

[107] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 71–85, 2016.

[108] Janakiram MSV. How firecracker is going to set modern infrastructure on fire. https://thenewstack.io/how-firecracker-is-going-to-set-modern-infrastructure-on-fire/, December 2018.

[109] Janakiram MSV. Aws, microsoft, and google agree the session is the new unit of compute. they disagree on how to isolate it. https://thenewstack.io/agent-session-aware-runtime/, 2026.

[110] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, page 29, 2003.

[111] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP'12, page 126–142.

[112] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018.

[113] Open Container Initiative. Open container initiative runtime specification. https://github.com/opencontainers/runtime-spec, 2026.

[114] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Using Trāṭr to tame adversarial synchronization. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3897–3916, August 2022.

[115] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: large language model connected with massive apis. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, 2024.

[116] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[117] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 291–304, 2011.

[118] Prathyush PV. klockstat: An ebpf tool to monitor linux kernel lock contentions. `https://prathyushpv.github.io/2019/04/30/kLockStat.html`, August 2019.

[119] Brandon Radosevich and John Halloran. Mcp safety audit: Llms with the model context protocol allow major security exploits. https://arxiv.org/abs/2504.03767, 2025.

[120] Allison Randal. The ideal versus the real: Revisiting the history of virtual machines and containers. *ACM Comput. Surv.*, 53(1), feb 2020.

[121] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. BlockFlex: Enabling storage harvesting with Software-Defined flash in modern cloud platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 17–33, July 2022.

[122] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, July 1974.

[123] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. *SIGARCH Comput. Archit. News*, 39(3):57–68, jun 2011.

[124] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, nov 1997.

[125] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, 2023.

[126] Amd sev-snp: Strengthening vm isolation with integrity protection and more. `https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf`, 2020.

[127] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, July 2020.

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

[129] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[130] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for Multi-Tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, October 2012.

[131] Jeremiah Spradlin and Zach Koopmans. gvisor security basics - part 1. `https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/`, November 2019.

[132] Vasily Tarasov, Erez Zadok, , and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *Usenix ;login*, 41(1), 2016.

[133] Intel® trust domain extensions (intel® tdx). `https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html`, 2023.

[134] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[135] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, 2021.

[136] What is v8? `https://v8.dev/`.

[137] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 281–292, 2012.

[138] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 181–192, 1998.

[139] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[140] Scott Votke, Seyyed Ahmad Javadi, and Anshul Gandhi. Modeling and analysis of performance under interference in the cloud. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 232–243, 2017.

[141] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, 2018.

[142] Xu Wang. Kata Containers and gVisor: a Quantitative Comparison. `https://www.openstack.org/summit/berlin-2018/summit-schedule/events/22097/kata-containers-and-gvisor-a-quantitative-comparison`, November 2018.

[143] Webassembly. `https://webassembly.org/`, 2024.

[144] David Wiesen. Building a safe, effective sandbox to enable codex on windows. https://openai.com/index/building-codex-windows-sandbox/, 2026.

[145] Wikipedia. Non-blocking algorithm. `https://en.wikipedia.org/wiki/Non-blocking_algorithm`, 2024.

[146] Wikipedia. Spearman's rank correlation coefficient. https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient, 2026.

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

[148] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. IsolateGPT: An execution isolation architecture for LLM-based agentic systems. In *Network and Distributed System Security Symposium (NDSS)*, 2025.

[149] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. https://arxiv.org/abs/2210.03629, 2023.

[150] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gvisor case study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, pages 16–16, 2019.

[151] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 379–391, 2013.

[152] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 771–788, 2023.

[153] Yusheng Zheng, Jiakun Fan, Quanzhi Fu, Yiwei Yang, Wei Zhang, and Andi Quinn. Agentcgroup: Understanding and controlling os resources of ai agents. https://arxiv.org/abs/2602.09345, 2026.