Container-Level Auditing in Container Orchestrators with eBPF

Fábio Junior Bertinatto, Daniel Arioza, Jéferson Campos Nobre, and Lisandro Zambenedetti Granville

Abstract This paper examines the application of eBPF (extended Berkeley Packet Filter) for achieving more precise auditing at the container level in container orchestrators such as Kubernetes. We address the challenges associated with auditing container behavior and highlight the advantages of leveraging eBPF to monitor container activities at the kernel level. We propose an eBPF-based solution that enhances transparency with respect to operations performed within containers. Overall, this study suggests that the use of eBPF for container-level auditing can provide valuable insights into container behavior and improve the security of containerized applications.

1 Introduction

The use of Linux containers has experienced a significant surge in popularity in recent years. This popularity comes from their ability to package and isolate applications in a portable manner. A survey conducted by the Cloud Native Computing Foundation (CNCF) in 2022 [1] revealed that 44% of the respondents reported using containers

Fábio Junior Bertinatto
Institute of Informatics - Federal University of Rio Grande do Sul, Porto Alegre, Brazil, e-mail: fabio.bertinatto@inf.ufrgs.br

Daniel Arioza
Institute of Informatics - Federal University of Rio Grande do Sul, Porto Alegre, Brazil, e-mail: daniel.almeida@inf.ufrgs.br

Jéferson Campos Nobre
Institute of Informatics - Federal University of Rio Grande do Sul, Porto Alegre, Brazil, e-mail: jcnobre@inf.ufrgs.br

Lisandro Zambenedetti Granville
Institute of Informatics - Federal University of Rio Grande do Sul, Porto Alegre, Brazil, e-mail: granville@inf.ufrgs.br

for most applications within their organizations. Furthermore, 35% of the respondents used containers for a few production systems, while 9% were actively evaluating the technology. According to recent data [2], Kubernetes[1], the leading container orchestration platform, is used by nearly half of all organizations as their primary tool for deploying and managing applications

This surge in containerized applications and Kubernetes usage has introduced challenges, especially in real-time auditing for cluster administrators and network operators. Traditional methods, like executing a shell interpreter within the container, do not persist commands, leaving no audit trail post-termination. Kubernetes offers an *Events* mechanism for cluster events but lacks the capability to record shell interpreter commands.

Addressing this gap, our paper introduces an eBPF-based auditing solution. We employ eBPF for kernel instrumentation to capture commands executed within Bash shell interpreters by targeting the *readline* function. Our approach integrates an eBPF program loader service within containers, creating Kubernetes *Events* resources. This service provides cluster administrators with a detailed record of executed commands, enhancing the ability to audit and troubleshoot containerized applications.

The paper is organized as follows: Section 2 covers the fundamentals of Linux containers, orchestrators, and eBPF. Section 3 reviews existing approaches using eBPF for container monitoring and auditing. Our proposed methodology and its assessment are detailed in Section 4. Section 5 concludes with final thoughts and potential future research directions.

2 Background

Containers have gained prominence as a viable option for deploying applications at scale, primarily due to their isolation capabilities and the ease with which applications can be packaged and deployed. As the adoption of containers increased, container orchestrators emerged as a solution for managing large-scale deployments of containerized applications. Simultaneously, eBPF has gained popularity as an efficient and flexible system monitoring solution. In this section, we explore these technologies and examine their key features to better understand how they can complement each other.

2.1 Containerization

Containers are isolated processes on a host machine, crucial in Linux systems for achieving process-level isolation through *namespaces*, *cGroups*, and *seccomp*. These

[1] https://kubernetes.io

kernel features isolate resources, control resource access, and restrict system calls, enhancing security and efficiency [3, 4].

Fig. 1 illustrates the need for an additional management interface to incorporate the aforementioned Linux kernel concepts into containers. The subsequent subsections will provide further details regarding this interface.

Container runtimes, like *runc* and *crun*, are command-line tools that manage containers in accordance with Open Container Initiative (OCI) specifications. They interact with container engines that handle additional functionalities like image retrieval and mounting [5].

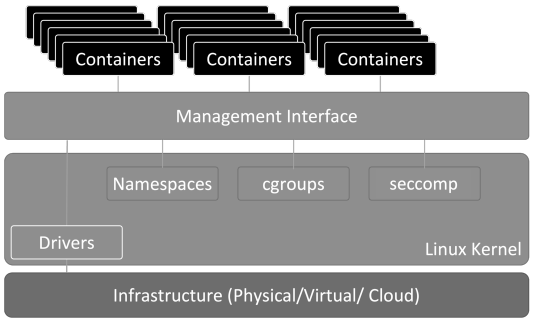

Fig. 1 Architecture of Linux Containers.

subsectionContainer Orchestrators

Automated management of containerized services is vital due to the complexity of handling numerous containers. Orchestration frameworks, particularly Kubernetes, have become essential in managing container ecosystems at scale, addressing resource heterogeneity and environmental constraints [6].

2.1.1 Kubernetes

Kubernetes, a project initially developed by Google and now maintained by the CNCF, is a prominent orchestrator. It interfaces with various container engines via the Container Runtime Interface (CRI), ensuring compatibility across different container management tools [2, 7].

Kubernetes objects like *Pods* represent the cluster's intended state, with each object playing a specific role in the system's overall functionality. The orchestration process involves managing these objects to maintain desired states [8].

2.2 eBPF

Extended BPF (eBPF), evolving from BPF, is a critical Linux kernel technology for system monitoring and tracing. It facilitates kernel-level program execution in response to events, making it a powerful tool for container auditing and security [9–13].

eBPF's versatility for applications like program tracing and performance analysis is particularly useful for container monitoring, providing deep insights into operations. Programs written in C-like syntax are compiled, loaded, and executed in the kernel, offering real-time system operation insights [14, 15].

The programmability of eBPF allows for innovative container auditing solutions, enabling tracking of container actions for enhanced security and observability. This adaptability makes eBPF crucial in modern Linux environments for applications requiring detailed observability, such as container-level auditing [16].

3 Related Work

In the domain of container auditing and monitoring, recent studies have primarily focused on various aspects of performance monitoring, security, and network analysis, often leveraging eBPF technology. Our work aims to extend these concepts to the specific challenge of auditing commands issued by cluster administrators in containerized environments, an area that has not been thoroughly explored in existing literature.

Cassagnes et al. [6] discuss the application of eBPF for system performance monitoring in computer systems, underscoring the limitations of traditional methods. Their work highlights eBPF's ability to collect kernel-level data efficiently, which is relevant for our approach to monitor administrative actions in containers.

Nam et al. [17] explore eBPF's role in enhancing inter-container communication security. Their focus on security and unauthorized access prevention aligns with our goal of ensuring that administrative actions within containers are secure and traceable.

Liu et al. [18] present a system that uses eBPF for analyzing network traffic in dynamic containerized networks. Their comprehensive approach to network observability demonstrates the feasibility of using eBPF for in-depth monitoring in complex container environments, which is analogous to our focus on detailed command auditing.

While Burns [19] introduces the sidecar pattern for observability in Kubernetes, this method presents limitations such as resource overhead and the need for manual instrumentation. Our work aims to overcome these limitations by proposing a more integrated and automated approach to auditing within container orchestration environments.

Rice [20] suggests using a single eBPF-based agent per node, moving away from the sidecar pattern. The Tracee forensics tool, based on this concept, uses eBPF to

trace activities on the host OS, including containerized applications. This approach to collecting and analyzing runtime events offers insights into implementing an efficient container-level auditing system.

To address the gap in existing literature, our work focuses on developing a method for accurately identifying and auditing specific commands executed by cluster administrators inside containers, leveraging the strengths of eBPF for real-time monitoring and security enhancement.

4 Evaluation

In this section, we assess the viability of employing eBPF programs within containers for monitoring and auditing objectives. In 4.1, we expound upon the eBPF program employed to capture executed commands within a container and elucidate our methodology for integrating it with Kubernetes. Lastly, we present the outcomes of our experiments in 4.3.

4.1 Propose and Implementation

In our implementation, we utilized an eBPF program to capture the commands executed within the Bash shell interpreter. This was achieved by instrumenting the *readline*[2] function, which is used by Bash to read user-provided commands. Our focus on Bash stems from its status as the default shell in most contemporary Linux distributions.

The eBPF program we employed monitored commands executed by any user on the system. While this may pose challenges in systems with multiple users, it becomes advantageous within the context of Linux containers. Containers have a limited perception of the system, and the program running inside a container perceives the container itself as the entire system. Consequently, our eBPF program captures commands executed by any process running within the same container.

However, running eBPF programs in containers introduces additional complexities. Notably, one major challenge is ensuring that the eBPF program runs within the same namespace as the process being monitored. Specifically, our eBPF program needs to run within the same mount and *PID* namespaces as the shell interpreter. To address this, we employed the *nsenter*[3] tool, which allows the execution of a program within specified namespaces. We ran the eBPF program using nsenter to ensure it runs in the appropriate namespaces.

To initiate the eBPF program as soon as a shell interpreter starts running in a container, we introduced two additional components to our solution: a runc wrapper

[2] https://git.savannah.gnu.org/cgit/readline.git/tree/readline.c?h=readline-8.2#n351

[3] https://man7.org/linux/man-pages/man1/nsenter.1.html

and a service running on the worker node alongside the *kubelet*. The *runc* wrapper detects the PID of the Bash process within any container and sends this information to the service running on the worker node. The service accepts incoming PIDs via an HTTP interface and runs the eBPF program within the specified PID's namespace using the nsenter tool.

Once the eBPF program is running within a container, it captures all commands executed within any Bash process running in that container. It communicates this information back to our service running on the host machine. This communication is facilitated by a Unix Domain Socket, which is mapped within the container and exposes the service's HTTP interface. When the service receives information from the eBPF program, it creates corresponding Events resources within the Kubernetes API server. As a result, Kubernetes cluster administrators gain a comprehensive view of the actions executed within the container.

Figure 2 illustrates the workflow of our implementation.

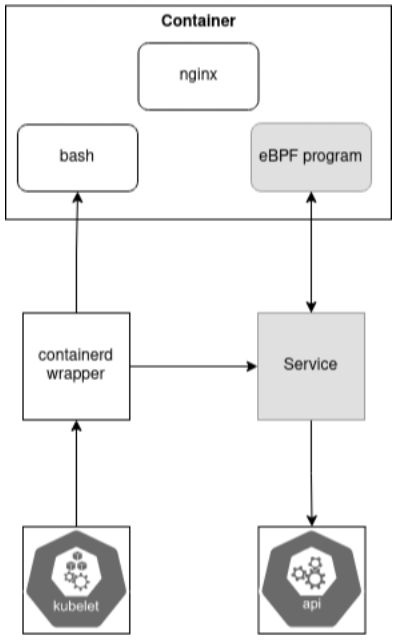

Fig. 2 Architecture of our implementation.

Our solution requires manual configuration. Specifically, a configuration change is needed in Containerd to invoke our wrapper instead of directly calling Runc. The excerpt in Listing 1 provides additional configuration details that should be added to the */etc/containerd/config.toml* file on the worker node, specifically under the *[plugins."io.containerd.grpc.v1.cri".containerd.runtimes]* section.

To enable applications running in Kubernetes to utilize our runc wrapper and, as a result, leverage our entire solution, it is necessary to create a RuntimeClass and

```
[plugins."io.containerd.grpc.v1.cri".containerd.
↪  runtimes.wrapper]
  runtime_type = "io.containerd.runc.v1"
  pod_annotations = ["*"]
  container_annotations = ["*"]

[plugins."io.containerd.grpc.v1.cri".containerd.
↪  runtimes.wrapper.options]
  BinaryName="/usr/bin/wrapper"
```

Listing 1: Containerd configuration.

reference it in the workloads that require monitoring. Listing 2 illustrates the YAML file that can be used to create this resource in Kubernetes.

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: my-wrapper-name
handler: wrapper
```

Listing 2: RuntimeClass that uses our runc wrapper.

Lastly, it is necessary to specify the designated *RuntimeClass* in the deployment object of the containerized application that requires monitoring. For example, Listing 3 demonstrates a *Deployment* object that utilizes the *RuntimeClass* defined in Listing 2. Additionally, this object includes a mapping that allows the eBPF program to be accessed from the host's directory and executed by our service on the worker node via *nsenter*.

4.2 Environment

In order to assess the viability of using eBPF programs within containers, we conducted experiments in a test environment that emulates a typical deployment of containerized applications. Our test environment comprised a Kubernetes 1.26 cluster with a single node, running on a virtual machine. Opting for a single-node Kubernetes setup allowed us to concentrate on the feasibility of our approach rather than the scalability or performance of the system. The Kubernetes cluster was set up using the *hack/local-up-cluster.sh* script, which is available in the Kubernetes source code.

The virtual machine was provisioned using Vagrant version 2.2.9 and utilized the Kernel Virtual Machine (KVM) virtualization technology on a Linux-based system.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      runtimeClassName: wrapper
      containers:
      - name: nginx
        image: nginx:latest
        volumeMounts:
        - mountPath: /ebpf
          name: ebpf-program-mount-point
      volumes:
      - name: ebpf-program-mount-point
        hostPath:
          # Location on the host where
          # the eBPF program is located
          path: /path-on-the-host
```

Listing 3: Deployment Object using the custom wrapper RuntimeClass

The virtual machine was equipped with 4 GiB of Random Access Memory (RAM) and an Intel Skylake CPU with 4 cores. It ran Fedora Linux version 37 with a Linux kernel version 6.2.8.

The host machine employed for running the virtual machine was a ThinkPad P1 Gen 3 laptop, equipped with 32 GiB of RAM and an Intel i7-10850H processor featuring 12 cores. The host machine ran Fedora Linux version 37 with a Linux kernel version 6.2.8.

4.3 Experiments

After completing the manual steps outlined in subsection 4.1, we proceeded with a series of experiments to evaluate the effectiveness of our solution. To simulate a real-world troubleshooting scenario, we initiated a Bash shell interpreter within the running container *nginx* from Listing 3 and executed multiple commands.

These commands were designed to replicate a troubleshooting scenario where an administrator investigates the underlying cause of stalled requests within an *nginx*

worker process. During this process, the administrator examines the *nginx* logs, monitors active processes, and utilizes the strace tool[4] to trace the specific system call where the issue arises.

During the experimentation process, our service successfully generated Kubernetes *Event* objects for each command executed within the container. Table 1 provides a comprehensive list of the *Events* created in our Kubernetes cluster throughout the course of our experiments. The data presented in the table was obtained using the *kubectl* command, filtering out any details unrelated to our specific experiments.

Table 1 Kubernetes Events.

Object	**Message**
pod/nginx-deployment-89c6ff86b-92ndx	ls
pod/nginx-deployment-89c6ff86b-92ndx	ip a
pod/nginx-deployment-89c6ff86b-92ndx	vim /etc/nginx/nginx.conf
pod/nginx-deployment-89c6ff86b-92ndx	cat /var/log/nginx/access.log
pod/nginx-deployment-89c6ff86b-92ndx	cat /var/log/nginx/error.log
pod/nginx-deployment-89c6ff86b-92ndx	ps
pod/nginx-deployment-89c6ff86b-92ndx	apt update
pod/nginx-deployment-89c6ff86b-92ndx	apt install procps
pod/nginx-deployment-89c6ff86b-92ndx	ps
pod/nginx-deployment-89c6ff86b-92ndx	strace -p 1
pod/nginx-deployment-89c6ff86b-92ndx	apt install strace
pod/nginx-deployment-89c6ff86b-92ndx	strace -p 1
pod/nginx-deployment-89c6ff86b-92ndx	strace -p 28

4.4 Comparison of Container-Level Monitoring and Auditing Solutions

In order to validate our solution, we conducted a comprehensive comparison with established tools in the market, adjusting their configurations to encompass a scope similar to ours. In this subsection we describe each of the compared solutions and provide details on the methodology and evaluation criteria used.

Falco is a real-time security monitoring tool designed to identify anomalous behaviors in applications and infrastructure. Tracee, another contender in container monitoring and security, excels at capturing and analyzing system events within containers, providing insights into container-level activities. Finally, Auditd, an established auditing solution, was included in our analysis. Auditd offers detailed system-level auditing capabilities.We compare our eBPF solution in terms of resource usage and responsiveness, justifying each comparison.

Specific rules were defined for each solution to capture and evaluate relevant container events. For Falco and Trace, we configured rules like ”Terminal in Container”

[4] https://strace.io/

and ”Write below etc.” to detect suspicious activities. Auditd, being an established solution, required less rule customization, showcasing the adaptability of the monitoring solutions.

Performance tests were conducted using a custom scrip that executed two specific test cases: ’Terminal in Container’ and ’Write below etc.’ For the ’Terminal in Container’ test case, it simulated the use of a shell in a container. For the ’Write below etc.’ test case, it detected attempts to write to any file below the ’/etc’ directory. The script was executed with the following parameters:

- **Repetitions:** 100,000 times
- **Resource Sampling Interval:** 0.1 second

The performance results are displayed in Table 2, which presents a comparison of three distinct metrics. The first column concerning the comparison of the response times, the second column regarding the CPU usage, and the third column pertaining to the assessment of memory usage. There are also graphs representing some level of the performance of the tools hence making it easier to analyze in very clear and comprehensive graphs. The evaluation comprised three rounds of tests: specifically, Test Type A was executed in a Terminal inside a Container and Test Type B that consumed many Write operations and others.

Table 2 Comparative Analysis of Response Time, CPU Usage, and Memory Usage

Test	Tool	Response Time (s)	CPU Usage (%)	Memory Usage (MB)
Test 1A	Falco	49.16	-0.60	6.43
	Tracee	50.12	0.20	4.77
	eBPF	49.12	-0.50	6.30
	Auditd	49.34	-0.40	6.20
Test 2B	Falco	49.44	0.40	1.55
	Tracee	49.89	0.80	3.98
	eBPF	49.33	0.30	1.50
	Auditd	49.42	0.20	1.40
Test 3A	Falco	49.35	0.00	2.34
	Tracee	50.02	0.80	4.26
	eBPF	49.53	-0.10	2.20
	Auditd	49.63	0.00	2.10
Test 4B	Falco	49.54	0.00	0.76
	Tracee	49.83	0.60	3.55
	eBPF	49.74	-0.10	0.70
	Auditd	49.79	0.00	0.60
Test 5A	Falco	49.71	0.00	2.34
	Tracee	49.52	0.80	4.26
	eBPF	49.42	-0.10	2.20
	Auditd	49.44	0.00	2.10
Test 6B	Falco	49.84	0.00	0.76
	Tracee	49.93	1.60	3.55
	eBPF	49.90	-0.10	0.70
	Auditd	50.00	0.00	0.60

We provide a concise analysis of memory, CPU utilization, and response time from our comparative study:

- **Our Solution:** Exhibited excellent efficiency in memory and CPU usage, leveraging eBPF's event-specific capture. It outperformed Auditd and matched Tracee and Falco in resource utilization and response time, offering real-time monitoring and immediate alerts for suspicious activities.
- **Tracee:** Showed low memory and CPU impact, suitable for constrained environments, with real-time response capabilities.
- **Auditd:** Demonstrated heavier resource usage under complex rules or high event volume. It introduces latency due to log writing to disk.
- **Falco:** Provided efficient container behavior monitoring, though resource usage can escalate with extensive rules.

We propose a solution to this challenge based on eBPF that strikes the balance between resource efficiency and immediate responsiveness, offering a cost-efficient alternative for auditing and monitoring of containers. It enhances security without really adding significant overheads, making it well suited for efficient paradigms preferred by container environments.

5 Conclusion

This solution combines security and efficacy to allow organizations in reinforcing their strategies for container security. The paper exhibits how effectively eBPF can effectively be employed in auditing tasks in Kubernetes orchestrators through capturing, as well as analyzing, the commands at a container level, basically the administrator instructions. Our eBPF-based solution flawlessly integrated within Kubernetes increases visibility, security, and observability over containerized systems delighting administrators with fine-grained insights. Experimental results carried out over synthetic Kubernetes clusters validate the efficacy of our approach in resource utilization while keeping approximately responsive the system.

Focusing on security and with a minimum system performance impact, our solution is perfect for any organizational setup looking for bulletproof container security solutions without having to compromise the system's performance. The comparison indicates the importance of memory, CPU utilization, and response time in ensuring that the security within the container environment is guaranteed. Our eBPF solution seems an effective option pointing towards its capability to optimize resources as well as its high potential to identify any future threat faster. The container-level monitoring tools offer the comparative analysis for memory, cpu usage and response with a goal of providing security for the container environment.

Looking ahead, enhancing the eBPF solution involves broadening support for various shell interpreters, evaluating its effectiveness in larger, multi-node Kubernetes clusters, and ensuring scalability under diverse workloads. Further, streamlining

deployment through automation techniques like Kubernetes *DaemonSet* will improve efficiency.

Acknowledgements This work was supported by The São Paulo Research Foundation (FAPESP) under the grant number 2020/05152-7, the PROFISSA project.

References

1. C. N. C. Foundation, "Cncf annual survey 2022," mar 2023. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022
2. I. CDatadog, "9 insights on real-world container usage," mar 2023. [Online]. Available: https://www.datadoghq.com/container-report
3. M. Zhan, Y. Li, H. Yang, G. Yu, B. Li, and W. Wang, "Runtime detection of application-layer cpu-exhaustion dos attacks in containers," *IEEE Transactions on Services Computing*, pp. 1–12, 2022.
4. J. Simonsson, L. Zhang, B. Morin, B. Baudry, and M. Monperrus, "Observability and chaos engineering on system calls for containerized applications in docker," *Future Generation Computer Systems*, vol. 122, pp. 117–129, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21001163
5. O. Specification, "About the open container initiative - open container initiative." [Online]. Available: https://opencontainers.org/about/overview/
6. C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," *Proceedings of IEEE/IFIP Network Operations and Management Symposium 2020: Management in the Age of Softwarization and Artificial Intelligence, NOMS 2020*, no. August 2019, 2020.
7. T. K. Authors, "Container runtime interface (cri)," feb 2023. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cri/
8. ——, "Pods," feb 2023. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/
9. S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," *Proceedings of the Winter 1993 USENIX Conference*, pp. 259–269, 1993.
10. T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," *CoNEXT 2018 - Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pp. 54–66, 2018.
11. M. Abranches, O. Michel, E. Keller, and S. Schmid, "Efficient network monitoring applications in the kernel with ebpf and xdp," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2021, pp. 28–34.
12. J. Schulist, D. Borkmann, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (bpf)," feb 2023. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/filter.txt
13. M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys*, vol. 51, no. 2, 2018.
14. eBPF.io Authors, "ebpf - introduction, tutorials & community resources," feb 2023. [Online]. Available: https://ebpf.io
15. M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3371038

16. R. Abranches, F. Tuma, R. Guimarães, and M. Vieira, "ebpf: a game-changer for network monitoring and security," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1815–1842, 2021.
17. J. Nam, S. Lee, P. Porras, V. Yegneswaran, and S. Shin, "Secure inter-container communications using xdp/ebpf," *IEEE/ACM Transactions on Networking*, pp. 1–14, 2022.
18. C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2020-Decem, pp. 697–702, 2020.
19. B. Burns, *Designing Distributed Systems*. O'Reilly Media, Inc., 2018.
20. L. Rice, *What Is eBPF?* O'Reilly Media, Inc., 2022.