Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информативна и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	Security-First OCI Runtime: Design and Comparative Evaluation with Sandboxed Runtimes Среда выполнения OCI, ориентированная на безопасность: проектирование и сравнительная оценка с использованием изолированных сред выполнения	**The implementation of a language server for the Jasmin cryptographic primitive description language** **Реализация языкового сервера для языка описания криптографических примитивов Jasmin**

Работу выполнил / Thesis is executed by		**Платонов Иван Васильевич**		подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis		**Садовых Андрей Александрович**		подпись / signature
Консультанты / Consultants		**Салтанов Кирилл Сергеевич**		подпись / signature

Иннополис, Innopolis, 2025

Contents

List of Tables

List of Figures

Abstract

The Open Container Initiative (OCI) specifications decouple image format, runtime, and tooling, enabling containers built with one implementation to be executed by another runtime and orchestrated via a third CLI. This modularity, together with the open-source nature of most components, allows deep customization of container execution to meet specific security requirements. Building on this flexibility, this project proposes and implements a security-first OCI runtime compatible with Docker and Podman. The runtime enforces strong defaults and automatically generates confinement policies (AppArmor and seccomp) during container setup. We will conduct a comparative study of security effectiveness and resource overhead versus sandboxed alternatives (gVisor and Kata Containers), assessing isolation strength, performance, and compatibility.

0.1 Introduction

0.1.1 Background

Containers have become the dominant deployment model for modern applications. However, unlike virtual machines that provide hardware-enforced isolation, containers share the host kernel with all other containers, creating a significant attack surface [1]. This shared kernel model means that a single kernel vulnerability can potentially compromise all containers on a host.

The Open Container Initiative (OCI) standardizes container runtimes. The reference implementation, runc, prioritizes performance and compatibility over security [2]. Linux provides kernel-level security mechanisms—seccomp for syscall filtering and AppArmor for mandatory access control—that can restrict container behavior [3]. However, these mechanisms require manual configuration, which is complex and time-consuming. Default configurations remain permissive to ensure maximum compatibility, leaving substantial attack surface exposed.

Alternative approaches like gVisor and Kata Containers provide stronger isolation by running containers in a user-space kernel or lightweight virtual machines. These approaches significantly reduce attack surface [4]. However, they introduce substantial performance overhead (20-100% degradation depending on workload) and operational complexity [5]. This creates a fundamental trade-off: stronger isolation comes at a performance cost.

0.1.2 Problem Statement and Research Gap

Three critical research gaps exist:

Gap 1: Missing Comparative Study. No recent research has compared a

properly hardened traditional runtime (runc with strict seccomp and AppArmor policies) against sandboxed alternatives. Existing comparisons use default runc configurations without security or old enough to reconduct them hardening [2], [4]. The actual security-performance trade-offs remain unknown.

Gap 2: No Developer-Friendly Automated Security. Current approaches require either manual policy configuration or integration into complex DevOps pipelines. Developers need a simple runtime that automatically generates and enforces security policies without manual intervention or infrastructure changes. This is fundamentally different from pipeline-based security automation—it must work at the runtime level itself [3].

Gap 3: Limited Automated Policy Generation. While prior work demonstrates seccomp profile generation through behavioral tracing [3], no comprehensive solution integrates seccomp, AppArmor, capability dropping, and device restrictions into a single runtime that developers can use directly, without manual configs editing [6].

0.1.3 Research Questions

This thesis addresses:

1. Can automatically generated seccomp and AppArmor policies achieve security comparable to sandboxed runtimes while maintaining performance closer to default runc?

2. How should a container runtime automatically discover and synthesize comprehensive security policies from application behavior?

3. What is the actual security-performance trade-off when comparing a properly hardened traditional runtime against sandboxed alternatives?

0.1.4 Proposed Approach

We propose implementing a security-first OCI runtime that:

- Automatically traces container behavior using eBPF during application startup and execution
- Synthesizes minimal seccomp profiles (allowlisting only required syscalls)
- Generates AppArmor policies restricting filesystem access to necessary paths
- Drops unnecessary Linux capabilities and restricts device access
- Requires no developer configuration or manual policy creation
- Integrates as a standard OCI runtime compatible with Docker and Kubernetes

The implementation will support both learning mode (behavioral profiling and policy generation) and enforcement mode (using generated policies).

0.1.5 Experimental Evaluation

We will compare security and performance across five configurations:

- **runc (default):** Standard OCI runtime with default policies
- **Proposed runtime:** Automated security generation and enforcement
- **gVisor:** User-space kernel sandbox approach
- **Kata Containers:** Virtual machine-based isolation

Evaluation will use representative workloads: web servers (nginx), databases (redis, postgresql), language runtimes, and microservices. Metrics will include startup time, CPU overhead, memory footprint, I/O performance, and security effectiveness against known container escape vectors [7].

0.1.6 Expected Contribution

This research will provide:

1. The first direct comparison of a properly hardened traditional runtime against sandboxed alternatives, providing empirical data on security-performance trade-offs

2. A practical, developer-friendly runtime that automatically generates and enforces security policies without manual configuration

3. Evidence-based guidance for organizations selecting appropriate container isolation strategies for different security requirements

If automated hardening can achieve security comparable to sandboxed solutions with significantly better performance, it represents a more practical security solution for the majority of deployments.

0.1.7 Thesis Organization

0.2 Literature Review

0.2.1 Container Architecture and Security Model

Containers provide operating system-level virtualization through Linux kernel mechanisms. Unlike virtual machines where each guest has its own kernel, containers share a single host kernel [1]. This architectural choice provides significant performance advantages but concentrates attack surface. Flauzac et al. [8] review native container security, noting that while containers impose minimal overhead compared to virtual machines, the shared kernel creates weaker isolation boundaries. A kernel exploit that succeeds against one container can potentially compromise all containers on the host.

The Open Container Initiative (OCI) defines standard container runtime specifications, enabling portability across platforms. Merkel [9] introduced Docker, which popularized container technology and drove adoption of these standards. Kozhirbayev and Sinnott [10] compare container performance against hypervisor-based approaches, confirming the performance advantage of containers at the cost of weaker isolation.

0.2.2 Linux Security Mechanisms

Seccomp (Secure Computing Mode). Seccomp enables syscall filtering at the kernel level using Berkeley Packet Filter rules. By restricting available syscalls, seccomp reduces the container attack surface regardless of other permissions granted. Lopes et al. [3] demonstrate that seccomp profiles can be automatically generated through runtime syscall tracing, reducing available syscalls by up to 80% compared to default configurations while maintaining application

compatibility. Schrammel et al. [11] extend syscall filtering with memory isolation techniques, showing how kernel-level mechanisms can effectively restrict container behavior.

Canella et al. [12] present automated approaches for generating strict syscall filters, combining static and dynamic analysis to identify all possible syscall requirements. Their work addresses limitations of purely behavioral profiling by incorporating techniques to capture infrequently executed code paths.

Mandatory Access Control. AppArmor provides path-based mandatory access control, confining programs to specific filesystem access patterns. Mattetti and Allouche [6] combine seccomp with AppArmor policies for comprehensive automatic security hardening. By integrating multiple enforcement mechanisms, organizations achieve stronger security than either mechanism alone.

eBPF-Based Monitoring. eBPF enables efficient kernel-level program execution for monitoring and enforcement. Aich et al. [13] demonstrate eBPF applications in security, showing how kernel-level tracing can provide fine-grained visibility into container behavior with minimal performance overhead.

0.2.3 Sandboxed Container Runtimes

Three primary approaches provide enhanced container isolation:

gVisor. gVisor implements a user-space kernel written in Go that intercepts syscalls before they reach the host kernel. By handling syscalls in an isolated sentry process, gVisor dramatically reduces the host kernel attack surface. Only a minimal set of syscalls required by gVisor itself reach the host kernel. However, this extra abstraction layer introduces substantial performance overhead.

Kata Containers. Kata Containers runs each container inside a lightweight virtual machine using hypervisors like QEMU or Firecracker. This approach

provides complete kernel isolation—each container receives its own dedicated guest kernel protected by hardware virtualization. This eliminates the shared kernel attack surface entirely.

runc. The default OCI runtime, runc, provides no isolation beyond standard kernel mechanisms (namespaces, cgroups, capabilities). It prioritizes performance and compatibility over enhanced security.

Comparative Analysis. Wang et al. [4] conduct performance and isolation analysis of all three approaches. Their evaluation shows that runc outperforms sandboxed alternatives in performance but offers weaker isolation. Importantly, they do not evaluate what security level can be achieved with properly configured policies on runc itself.

Viktorsson et al. [5] evaluate security-performance trade-offs across runtimes using representative microservice benchmarks. Their findings show that gVisor introduces 50-100% overhead, Kata introduces 20-40% overhead, and runc provides near-native performance. However, they use default runc configurations without hardened policies.

Espe et al. [14] provide comprehensive performance evaluation of container runtimes using detailed benchmarking frameworks. Their work establishes performance baselines but does not address security hardening scenarios.

Volpert et al. [2] provide empirical analysis of OCI runtime isolation capabilities using eBPF-based instrumentation. They measure CPU overhead, memory latency, I/O performance, and container lifecycle operations, establishing that runc delivers optimal performance with default configurations but do not compare against hardened configurations.

0.2.4 Automated Policy Generation and Container Hardening

Automated Seccomp Profiling. Lopes et al. [3] demonstrate that runtime tracing can generate application-specific seccomp profiles reducing attack surface substantially. Their evaluation shows that custom profiles improve security compared to defaults while maintaining compatibility.

Policy Generation Frameworks. Li et al. [15] demonstrate automatic generation of inter-service access control policies in microservices using static analysis. Their work shows that automated approaches can effectively constrain behavior based on observed requirements.

Container Security Hardening. Mattetti and Allouche [6] present comprehensive framework for automatic hardening combining multiple enforcement mechanisms. Pothula et al. [16] develop security control mapping for runtime container hardening.

Threat Modeling and Security Analysis. Wong et al. [17] conduct comprehensive threat modeling of containers using the STRIDE framework, identifying vulnerabilities across the entire container supply chain. Their work confirms that containers present multiple attack surfaces requiring defense-in-depth. Getahun [18] extends threat analysis through detailed examination of container security threats and mitigation strategies.

0.2.5 Research Gap

Current literature reveals three critical gaps:

Gap 1: Missing Comparative Study. Existing runtime comparisons use default runc configurations without security hardening [2], [4], [5]. No research compares a properly hardened traditional runtime against sandboxed alternatives,

leaving security-performance trade-offs unknown.

Gap 2: Lack of Developer-Focused Solutions. Automated policy generation techniques exist [3], but no integrated runtime solution provides automatic discovery, synthesis, and enforcement of comprehensive policies at the runtime level for direct developer use without infrastructure changes.

Gap 3: Limited Integration. While individual components exist (seccomp profiling, AppArmor policies, eBPF monitoring), comprehensive integration of these mechanisms into a single developer-friendly runtime remains unexplored.

This thesis addresses these gaps by implementing an automated security-first OCI runtime and conducting the first direct comparison of hardened traditional runtimes against sandboxed alternatives using representative workloads and comprehensive evaluation metrics.

Bibliography cited

[1] K. Wang et al., "Characterizing and optimizing kernel resource isolation for containerized applications," *Future Generation Computer Systems*, vol. 141, pp. 234–247, 2023.

[2] S. Volpert, S. Winkelhofer, S. Wesner, and J. Domaschka, "An empirical analysis of common oci runtimes' performance isolation capabilities," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ACM, 2024, pp. 60–70. DOI: 10.1145/3629526.3645044.

[3] N. Lopes, R. Martins, M. E. Correia, S. Serrano, and F. Nunes, "Container hardening through automated seccomp profiling," in *Proceedings of the 6th International Workshop on Container Technologies and Container Clouds*, ACM, 2020, pp. 31–36. DOI: 10.1145/3429885.3429966.

[4] X. Wang and D. Johannesson, "Performance and isolation analysis of runc, gvisor and kata containers runtimes," *Cluster Computing*, vol. 25, pp. 2363–2380, 2022. DOI: 10.1007/s10586-021-03517-8.

[5] W. Viktorsson, C. Klein, and J. Tordsson, "Security-performance trade-offs of kubernetes container runtimes," in *2020 IEEE MASCOTS*, IEEE, 2020, pp. 1–4.

[6] M. Mattetti and Y. Allouche, "Automatic security hardening and protection of linux containers," *Security and Privacy in Communication Systems*, 2020.

[7] National Vulnerability Database, *Cve-2019-5736: Runc container escape vulnerability*, 2019.

[8] O. Flauzac, C. Gonzalez, and F. Nolot, "A review of native container security for running applications," *International Journal of Computer Science and Engineering*, vol. 22, no. 3, pp. 187–204, 2020.

[9] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[10] Z. Kozhirbayev and R. Sinnott, "A performance comparison of container-based hypervisors," in *Proceedings of IEEE 10th International Conference on Cloud Computing*, 2017.

[11] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Securing syscalls for pku-based memory isolation systems," in *Proceedings of the 31st USENIX Security Symposium*, USENIX, 2022, pp. 2421–2438.

[12] C. Canella et al., "Automating seccomp filter generation for linux applications," in *Proceedings of ACM CCS 2021*, ACM, 2021, pp. 2137–2152.

[13] Aich et al., "Ebpf for high-performance networking and security," *International Journal of Scientific Research and Advanced Engineering*, 2021.

[14] L. Espe, A. Jindal, V. Podolskiy, and M. Gerndt, "Performance evaluation of container runtimes," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, SCITEPRESS, 2020, pp. 273–281.

[15] X. Li et al., "Automatic policy generation for inter-service access control in microservices," in *Proceedings of the 30th USENIX Security Symposium*, USENIX, 2021, pp. 2485–2501.

[16] D. R. Pothula, K. M. Kumar, and S. Kumar, "Run time container security hardening using a proposed model of security control map," in *2019 Global Conference for Advancement in Technology*, IEEE, 2019, pp. 1–6.

[17] A. Y. Wong, E. Getahun Chekole, M. Ochoa, and J. Zhou, "Threat modeling, attack analysis, and mitigation strategies for containers," *Computers & Security*, vol. 131, p. 103 130, 2023.

[18] E. Getahun Chekole et al., "Threat modeling and security analysis of containers," *arXiv preprint*, 2021.