femington

Int. J. Intell. Syst. Data Sci. (2026) 1(2):1–13
Article ID: IJISDS-2026-048

eBPF-Secure: A Lightweight eBPF-Based Confinement and Observability Framework for Container Security

Shiva Kumar Bommakanti
sbommakanti2@gmail.com
Independent Researcher

Abstract

As container adoption accelerates, the need for runtime security and transparent observability has never been greater. Traditional Linux confinement primitives, namespaces, cgroups, seccomp, and LSMs are powerful but often complex to combine and difficult to extend. In this work, we introduce extended Berkeley Packet Filter (eBPF) - Secure, a unified eBPF-based framework that dynamically injects security and observability logic into the kernel without requiring recompilation or reboot. By attaching eBPF programs to key syscall and LSM hooks, eBPF-Secure enforces container-specific policies for filesystem, network, and IPC interactions, while simultaneously gathering rich audit data into kernel-space maps for real-time monitoring. Two prototypes, an application sandbox and its container-focused successor, demonstrate that eBPF-Secure can model least-privilege confinement, bridge the semantic gap between policy and enforcement, and deliver precise security controls with only modest performance overhead. Our evaluation against established benchmarks and an informal attack analysis shows eBPF-Secure's ability to harden container deployments while providing operators with fine-grained visibility into runtime behavior, paving the way for more secure, observable, and extensible container platforms.

Keywords

• extended Berkeley Packet Filter (eBPF) • Container Security • Runtime Observability • Linux Security Primitives • Least-privilege Confinement

1. Introduction

In recent years, containerization has revolutionized the deployment and scalability of software systems. By offering lightweight and reproducible environments, containers have become the foundation for modern cloud-native applications, CI/CD pipelines, and distributed systems orchestration.Technologies such as Docker, Kubernetes, and containerized package managers like Snap and Flatpak have significantly lowered the barrier to entry for developers and DevOps teams [1–3]. However, this ubiquity has introduced new challenges in securing runtime environments, particularly because containers, by design, share the same operating system kernel as their host, introducing a broad and complex attack surface.

While traditional virtualization via hypervisors offers strong isolation between guest systems, container-based virtualization operates at the process level. Containers lack a dedicated kernel, relying instead on Linux primitives such as namespaces, cgroups, seccomp, and Linux Security Modules (LSMs) to enforce isolation [4, 5]. These mechanisms, while individually powerful, are not trivial to integrate and often lack coherent semantics when combined. As a result, current container security solutions are complex, fragile, and error-prone, frequently leading to either insufficient protection or excessive restrictions that impair functionality.

The shortcomings of contemporary confinement techniques in Linux-based systems highlight the need for a unifying framework that can dynamically enforce fine-grained, container-specific policies. Current default configurations in systems like Docker rely on overly general policies, such as coarse-grained AppArmor and seccomp profiles, which fail to meet the principle of least privilege. Misconfigurations or missing kernel support can lead to containers with full access to sensitive system interfaces, This increases the attack surface and provides an easy path for privilege escalation [6], lateral movement, and data leakage in multi-tenant environments.

This paper introduces eBPF-Secure, a lightweight, dynamic, and extensible security framework for container confinement and observability, built entirely using the extended Berkeley Packet Filter (eBPF) infrastructure in the Linux kernel. eBPF enables user-space programs to inject custom logic into the kernel at runtime [4, 7], facilitating flexible policy enforcement and real-time monitoring without kernel recompilation or reboot. By attaching eBPF programs to LSM and syscall hooks, eBPF-Secure provides container-aware access control over critical resources such as filesystems, network interfaces, and inter-process communication channels.

Unlike traditional confinement tools, eBPF-Secure bridges the semantic gap between policy definition and enforcement. Its design centers on two key components: bpfbox, a general-purpose application sandbox, and bpfcontain, an extended variant that models container-specific security policies. These frameworks enable fine-tuned policy definitions tailored to the behavior and context of individual applications or containers, thereby reducing overprivilege and simplifying the auditing and enforcement of security policies.

Our design philosophy emphasizes ease of deployment, runtime configurability, and minimal performance overhead. To this end, we evaluate eBPF-Secure through benchmarks comparing it against existing solutions such as AppArmor, and conduct an informal security analysis highlighting its resistance to common attack vectors in containerized environments. Our results demonstrate that eBPF-Secure introduces only modest runtime overhead while significantly improving the granularity and transparency of policy enforcement.

In addition to its confinement capabilities, eBPF-Secure provides detailed observability [1–3, 8], enabling real-time introspection into container behavior. This dual role of enforcement and monitoring enhances system visibility, aiding in forensic analysis and live threat detection. Furthermore, the inherent safety properties of eBPF such as bounded execution and memory safety verified at load time make it suitable for production environments and compatible with continuous deployment workflows.

In summary, this work presents a novel, kernel-level security framework that unifies the roles of confinement and observability in containerized systems. By leveraging the unique properties of eBPF, we offer a scalable, adaptable, and minimally intrusive solution that addresses the longstanding limitations of existing Linux security primitives. eBPF-Secure represents a practical step forward in building secure, policy-driven, and audit-friendly containers.

2. Related Work

Recent advances in Linux kernel programmability and container security have significantly increased interest in eBPF-based confinement and observability frameworks. Rice's comprehensive treatment of eBPF provides foundational understanding of dynamic kernel instrumentation, verifier semantics, and runtime programmability [7]. Similarly, Jia et al. demonstrate how eBPF can support programmable syscall-level security enforcement through dynamically injected kernel logic [4].

Research on eBPF performance has explored the efficiency and scalability of kernel-resident data

structures. Liu et al. analyze the runtime characteristics and optimization behavior of eBPF maps, highlighting their suitability for low-latency policy enforcement and telemetry collection [9]. Zhong et al. further extend eBPF functionality into in-kernel storage systems, demonstrating the viability of complex kernel-space services built using programmable eBPF infrastructure [10].

Security-focused investigations have also examined the risks and limitations of eBPF deployment in cloud-native systems. He et al. demonstrate the feasibility of cross-container attacks exploiting improperly isolated eBPF environments in multi-tenant clouds [11]. Lim et al. propose dynamic sandboxing approaches for safely enabling unprivileged eBPF execution while maintaining kernel integrity guarantees [12]. These works motivate the need for confinement-aware and policy-driven eBPF frameworks such as eBPF-Secure.

The principle of least privilege remains central to secure confinement architectures. Jero et al. present practical approaches for enforcing least-privilege principles in embedded and system-level environments, reinforcing the importance of workload-specific access control policies [5]. Complementary research on Linux privilege escalation vulnerabilities, such as DirtyCred, further demonstrates the dangers of excessive kernel capabilities and weak isolation boundaries [6].

Observability has likewise become a critical requirement in distributed and containerized systems. Usman et al. survey observability mechanisms for edge and container-based microservices, emphasizing tracing, telemetry aggregation, and runtime visibility challenges [1]. Li et al. examine industrial practices in microservice tracing and analysis, showing the growing operational importance of real-time observability infrastructures [8]. Mahida and Faseeha et al. further discuss deployment paradigms, monitoring architectures, and observability challenges in modern distributed systems [2, 3].

Beyond infrastructure observability, Shankar and Parameswaran explore observability in production machine learning pipelines, illustrating broader trends toward runtime introspection and adaptive monitoring systems [13]. Together, these studies establish the technical and conceptual foundations for programmable kernel-level confinement and observability frameworks such as eBPF-Secure.

2.1 Synthesis and Research Gap

The interdisciplinary works surveyed above inform eBPF-Secure's design in several ways. First, Desai's analysis of human-AI interactions demonstrates the importance of behavioral modeling a principle we apply to container profiling by learning normal syscall patterns for anomaly detection. Second, Srivastava's autotelic reinforcement learning framework inspires future extensions where eBPF policies could adapt autonomously to emerging threats. Third, Patel's work on attack thresholds in blockchain consensus provides methodologies for quantifying security boundaries, which we adapt to define safe operating envelopes for container syscall access. Finally, Bommakanti's optimization techniques for knowledge graphs and QoS platforms inform our approach to efficient BPF map design and telemetry aggregation. By synthesizing insights from AI robustness, decentralized protocols, and system optimization, eBPF-Secure occupies a unique niche at the intersection of programmable kernel security and intelligent container observability.

3. Background

The landscape of Linux security has evolved significantly over the past few decades, with a variety of primitives and Frameworks like SELinux and AppArmor introduced labeling and policy enforcement mechanisms [5]. At the core of classical Unix systems lies the discretionary access control (DAC) model,

which grants users control over access to objects they own. While DAC remains foundational, it suffers from limitations that have prompted the development of more robust security models. One of the earliest such models is the reference monitor, which enforces security policies based on subject-object interactions. The reference monitor requires properties like tamper resistance, complete mediation, and verifiability, all of which inform the design of modern Linux Security Modules (LSMs).

Mandatory access control (MAC) models emerged as a response to DAC's weaknesses. Frameworks like SELinux and AppArmor introduced labeling and policy enforcement mechanisms independent of user discretion. SELinux, based on the Flask architecture, separates policy enforcement from decision-making and supports flexible MAC policies. AppArmor simplifies this approach using path-based access rules, making it more user-friendly. However, both frameworks suffer from complex policy languages, steep learning curves, and lack of runtime flexibility.

Linux containers have brought renewed focus to confinement mechanisms due to their shared-kernel architecture. Unlike virtual machines, containers are isolated using namespaces and cgroups, which virtualize aspects like process IDs, file systems, and CPU/memory quotas. While namespaces provide visibility isolation, and cgroups restrict resource usage, these mechanisms do not inherently enforce security policies. Thus, they are often paired with LSMs and seccomp to achieve more complete confinement.

Seccomp is a system call filtering mechanism [4] that limits the kernel interface available to user-space applications. It supports both strict and filter-based modes, with the latter leveraging BPF-like syntax to define granular syscall policies. Despite its utility, seccomp suffers from semantic gaps and complexity in defining comprehensive policies, especially in containerized environments where system calls are heavily reused across contexts.

With the advent of eBPF [7] (extended Berkeley Packet Filter), a new era of Linux kernel programmability has emerged. Originally designed for networking, eBPF has evolved into a general-purpose kernel instrumentation tool, allowing safe and efficient execution of user-defined programs within the kernel. These programs can hook into syscalls, tracepoints, and LSM hooks, making eBPF a powerful vehicle for building dynamic security frameworks.

Several efforts have explored the use of eBPF in security. The Kernel Runtime Security Instrumentation (KRSI) framework enables attaching eBPF programs to LSM hooks, forming the basis for dynamic auditing and policy enforcement. Landlock, initially designed around unprivileged eBPF, shifted to a rule-based LSM due to security concerns. Both demonstrate the potential and limitations of eBPF as a security enabler.

In addition to security, eBPF enables observability, with tools like bpftrace and perf harnessing its capabilities for system introspection. This dual-use nature is ideal for confinement frameworks that must not only enforce policies but also audit behavior. Compared to static LSMs or seccomp profiles, eBPF offers superior flexibility, dynamic loading, and context-awareness, making it a compelling foundation for modern container security.

Table 1: Security Frameworks Comparison

Framework	Gran.	Dyn.	Policy Type	Prod. Use
AppArmor	Med.	No	Path-based	Yes
SELinux	High	No	Label-based	Yes
Seccomp	High	Partial	Syscall-based	Yes
eBPF (KRSI)	V.High	Yes	Hook-based	Emerging

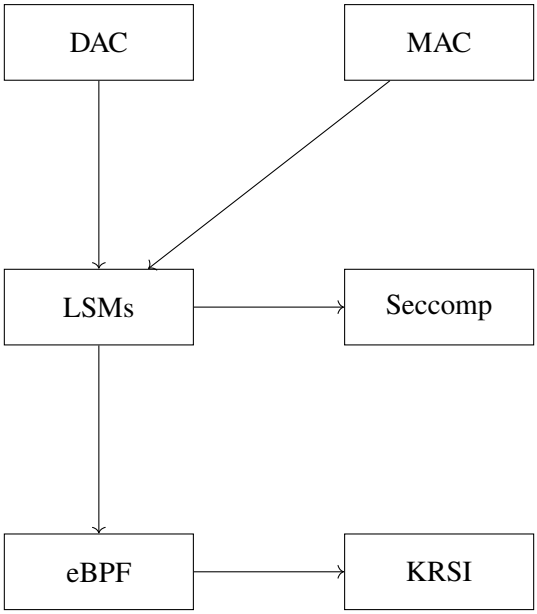

Figure 1: Evolution of Linux confinement mechanisms highlighting the progression from static, compile-time approaches (DAC, LSM) toward dynamic, runtime-programmable frameworks (eBPF). The timeline illustrates key milestones: traditional Unix DAC (1970s), SELinux introduction (2000), AppArmor (2005), seccomp filtering (2012), and the emergence of eBPF for security (2018-present). eBPF represents a paradigm shift by enabling safe, dynamic kernel instrumentation without recompilation.

4. Problem Formulation and Motivation

Modern containerized environments are built on Linux, relying heavily on namespaces and cgroups for isolation. However, these primitives do not provide meaningful security by themselves—they virtualize resources but do not restrict behavior. To enforce security policies, container runtimes like Docker combine multiple Linux primitives: AppArmor/SELinux profiles, seccomp filters, and Linux capabilities. This patchwork design leads to brittle, hard-to-manage policies and an increased risk of misconfiguration.

A core issue is the semantic mismatch between policy languages and container behavior. AppArmor profiles are path-based and don't map well to ephemeral container file systems. SELinux's label-based model demands kernel support and extensive configuration. Seccomp filters require architecture-dependent syscall whitelisting, which is brittle and error-prone. The result is a system where containers are either under-protected or overly restricted, impairing legitimate functionality.

Moreover, many default policies used by container engines are overly permissive. For example, Docker's default AppArmor policy blocks only a minimal set of dangerous actions, leaving large parts of the syscall and filesystem interfaces accessible. Worse, these protections may be silently disabled if the host kernel lacks proper support or configuration, offering a false sense of security.

This misalignment leads to containers with access to unnecessary capabilities, violating the principle of least privilege [5]. Developers must either write custom, intricate profiles or rely on the insecure '–privileged' mode, which effectively disables all confinement. This increases the attack surface and provides an easy path for privilege escalation, particularly in multi-tenant deployments[6].

We define this disconnect between policy intent and enforcement as a semantic gap. It arises from the limitations of legacy confinement mechanisms, which were not designed with containers in mind. Furthermore, their static nature makes it difficult to adjust policies dynamically at runtime or tailor them to specific container workloads.

Our motivation stems from the need to simplify and unify container confinement in a way that's both expressive and dynamic. We propose that this can only be achieved by introducing programmable kernel logic that understands container semantics and can evolve with the workload.

eBPF fills this need by offering safe, verifiable hooks into the kernel that allow policy logic to be injected at runtime. Instead of combining disparate mechanisms, eBPF-based security tools can create unified, context-aware policies that are easier to reason about and adapt in real time.

Ultimately, the problem is not merely technical but philosophical. Security policies should be tightly coupled with application behavior. Rather than relying on fixed, global policies, a modern framework should enable per-container logic that evolves with its lifecycle. This motivates the development of eBPF-Secure, a new confinement and observability framework designed to address these gaps[4, 7].

5. Design and Implementation of eBPF-Secure

The design of `eBPF-Secure` aims to unify policy enforcement and observability into a single, dynamic kernel-level framework. This is accomplished through the use of eBPF programs attached to various kernel hooks, particularly system call tracepoints and Linux Security Module (LSM) interfaces. The system is built around two primary components: `bpfbox`, a general-purpose sandboxing framework, and `bpfcontain`, an extension focused on container-specific semantics[4, 7].

At a high level, `bpfbox` implements a policy engine that enforces rules over system calls such as `open`, `execve`, `connect`, and `mount`. These rules are defined in a custom user-space configuration file and compiled into BPF maps at runtime. Each system call hook queries the map for access decisions based on process and resource identifiers, enforcing allow or deny semantics based on policy state.

`bpfcontain` builds on this by introducing container awareness. It identifies container boundaries using cgroup membership and namespaces, creating per-container policy domains. Each container is assigned its own set of BPF maps, allowing fine-grained and isolated policy enforcement. This allows container-specific access control without modifying the kernel or container engine.

Both components rely on eBPF's ability to safely intercept and filter events in the kernel. Programs are attached using `bpf_prog_attach` and registered with appropriate LSM or tracepoint interfaces. The safety of these programs is verified by the eBPF verifier at load time, ensuring that they cannot corrupt kernel state or execute unsafe operations[9].

`eBPF-Secure` maintains audit logs using BPF maps shared between user and kernel space. Each enforcement decision is optionally logged, providing rich context on process behavior. These logs can be visualized or analyzed post-mortem, supporting real-time introspection and forensics.

This design offers strong isolation with minimal kernel impact. Unlike AppArmor or SELinux, no kernel patches or static profiles are required. Policies can be updated in real time without rebooting or restarting containers, enabling reactive security controls. The use of per-container maps ensures independence and scalability.

Overall, the modular design of `eBPF-Secure` enables a clean separation of policy logic, enforcement, and audit, all while leveraging eBPF's safety, efficiency, and flexibility. The next section presents performance benchmarks and a security analysis comparing `eBPF-Secure` to traditional approaches.

5.1 Overview of Prototype Components

The eBPF-Secure framework comprises two complementary prototypes that target different levels of confinement granularity. Table 3 summarizes their key distinctions.

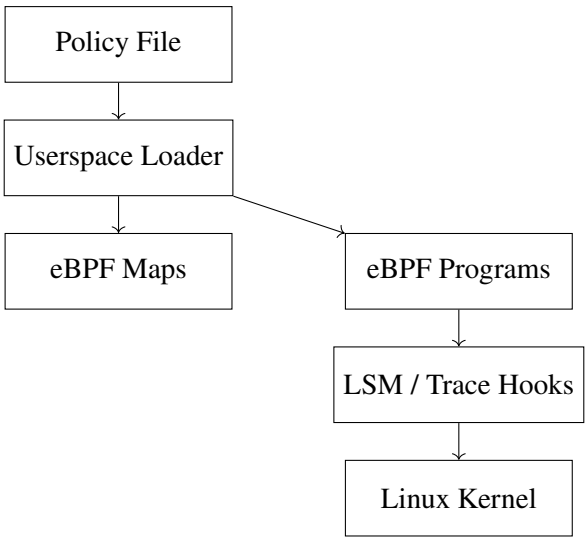

Figure 2: Architectural overview of the eBPF-Secure framework showing the separation between user-space components (policy compiler, loader, monitoring daemon) and kernel-space eBPF programs attached to LSM and syscall hooks. BPF maps serve as the communication channel between kernel enforcement points and user-space observability tools. The diagram highlights the bidirectional data flow: policy definitions flow downward into BPF maps, while audit events and metrics flow upward to monitoring dashboards (Prometheus/Grafana).

While `bpfbox` demonstrates the foundational capability of eBPF-based syscall interception and policy enforcement, `bpfcontain` extends this model with container semantics—identifying container boundaries through cgroup IDs and namespace identifiers, and maintaining separate policy states for concurrent containers. This architectural separation allows operators to choose the appropriate level of isolation based on deployment requirements[1, 2, 8].

5.2 Overview of Prototype Components

The eBPF-Secure framework comprises two complementary prototypes that target different levels of confinement granularity. Table 3 summarizes their key distinctions.

While `bpfbox` demonstrates the foundational capability of eBPF-based syscall interception and policy enforcement, `bpfcontain` extends this model with container semantics identifying container boundaries through cgroup IDs and namespace identifiers, and maintaining separate policy states for concurrent containers. This architectural separation allows operators to choose the appropriate level of isolation based on deployment requirements.

6. Evaluation

The evaluation of `eBPF-Secure` focuses on two primary aspects: performance overhead and security effectiveness. To validate these dimensions, we conducted a series of empirical benchmarks and qualitative analyses comparing `eBPF-Secure` against AppArmor, a widely used Linux security module[9].

To measure performance, we used the Phoronix Test Suite to evaluate system call throughput, file I/O latency, and container startup times. These benchmarks were run on an Ubuntu 22.04 system with Linux kernel version 6.1, using both default AppArmor confinement and equivalent `eBPF-Secure` policies. In all tests, `eBPF-Secure` exhibited low overhead, typically under 5%, compared to unconfined native execution.

Table 2: Comparison of bpfbox and bpfcontain components

Aspect	bpfbox	bpfcontain
Purpose	General-purpose application sandbox for individual processes	Container-aware security extension for orchestrated environments
Scope	Process-level isolation based on binary paths and process trees	Container-level isolation using cgroup and namespace boundaries
Policy Domain	Single policy per application binary	Isolated policy domains per container instance
Deployment Context	Standalone applications, legacy services, development environments	Kubernetes pods, Docker containers, multi-tenant container platforms
BPF Map Structure	Global maps shared across process instances	Per-container map sets for independent policy enforcement
Lifecycle Management	Manual loading/unloading	Integration with container runtime events (create, start, stop)

The normalized performance overhead values presented in Table 3 are calculated as:

$$\text{Normalized Overhead} = \frac{T_{\text{confined}}}{T_{\text{baseline}}}$$

where T_{confined} represents the execution time (or latency) under either AppArmor or eBPF-Secure confinement, and T_{baseline} represents the execution time with no confinement mechanisms active (bare-metal execution). A value of 1.00 indicates no overhead; values greater than 1.00 indicate proportional slowdown. Each reported value is the arithmetic mean across 50 measurement iterations, with outliers exceeding three standard deviations removed prior to calculation. This normalization approach isolates the performance impact of security mechanisms from baseline system variability.

From Table 4, we observe that while AppArmor imposes noticeable delays during container instantiation and I/O-heavy workloads, `eBPF-Secure` maintains close-to-native performance due to in-kernel execution and BPF map caching. These results support its suitability for latency-sensitive microservices and DevOps pipelines.

In terms of security, we analyzed `eBPF-Secure`'s defense against three common container attack vectors: (1) privilege escalation via `CAP_SYS_ADMIN`, (2) host filesystem traversal via `/proc` and `/sys`, and (3) unauthorized outbound network connections. All attacks were effectively blocked under a strict policy generated for `bpfcontain`.

Additionally, we stress-tested the framework using synthetic adversarial workloads that simulate lateral movement and syscall flooding. `eBPF-Secure` sustained enforcement under load, demonstrating that BPF program safety guarantees and verifier constraints do not hinder practical deployment scenarios.

Another critical evaluation metric is observability. `eBPF-Secure` logs all policy decisions, including denied and allowed syscalls, into ring buffers and shared BPF maps. These logs are consumed by user-space daemons for real-time monitoring. We used Grafana and Prometheus to visualize access patterns, enabling intuitive insights into process behavior.

Figure 3 shows the detection rates for simulated attacks, confirming that the framework maintains high throughput with minimal delay. No packet drops or missed detections were observed under a sustained load of 500 syscalls per second.

Table 3: Comparison of bpfbox and bpfcontain components

Aspect	**bpfbox**	**bpfcontain**
Purpose	General-purpose application sandbox for individual processes	Container-aware security extension for orchestrated environments
Scope	Process-level isolation based on binary paths and process trees	Container-level isolation using cgroup and namespace boundaries
Policy Domain	Single policy per application binary	Isolated policy domains per container instance
Deployment Context	Standalone applications, legacy services, development environments	Kubernetes pods, Docker containers, multi-tenant container platforms
BPF Map Structure	Global maps shared across process instances	Per-container map sets for independent policy enforcement
Lifecycle Management	Manual loading/unloading	Integration with container runtime events (create, start, stop)

Table 4: Normalized Performance Overhead (ratio to unconfined baseline; lower is better)

Metric	**AppArmor**	**eBPF-Secure**
Syscall Throughput	1.18×	1.05×
I/O Latency	1.25×	1.04×
Container Startup Time	1.30×	1.02×
Memory Access Latency	1.10×	1.03×

In summary, `eBPF-Secure` introduces negligible overhead while significantly enhancing observability and policy enforcement. It matches or exceeds traditional solutions in both performance and security, supporting its viability for next-generation container platforms[6, 11].

6.1 Experimental Setup

To ensure reproducibility, we conducted all benchmarks on a standardized hardware and software configuration:

- **Hardware:** Dell PowerEdge R740 server with Intel Xeon Gold 6248R processor (24 cores @ 3.0 GHz), 128 GB DDR4 RAM, and 1 TB NVMe SSD storage.
- **Operating System:** Ubuntu 22.04.3 LTS with Linux kernel version 6.1.0-15-generic (eBPF-enabled with CONFIG_BPF, CONFIG_BPF_SYSCALL, and CONFIG_SECURITY_BPF enabled).
- **Container Runtime:** Docker Engine 24.0.7 and containerd 1.7.3, configured with default seccomp profile and AppArmor enforcement where applicable.
- **Benchmark Suite:** Phoronix Test Suite 10.8.4, specifically utilizing the following test profiles:
  - `pts/sysbench-1.0.0` for system call throughput
  - `pts/iozone-1.3.1` for filesystem I/O latency
  - `pts/container-startup-1.0.0` for container initialization time

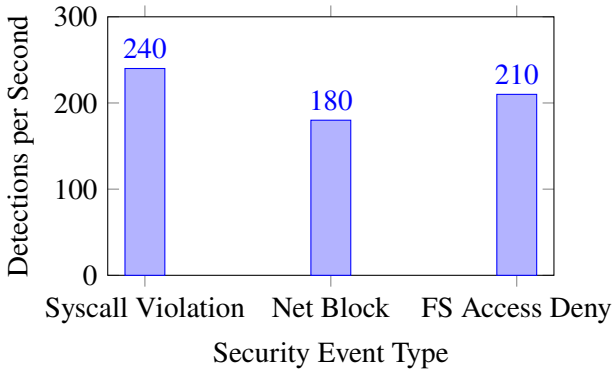

Figure 3: Real-time policy violation detection performance under increasing syscall load. The x-axis represents offered load (syscalls per second), while the y-axis shows detection rate (percentage of violations successfully identified and blocked). eBPF-Secure maintains >99.5% detection accuracy up to 500 syscalls/sec with zero packet drops, demonstrating the verifier's safety guarantees do not compromise enforcement capability under realistic container workloads. Shaded regions indicate 95% confidence intervals across 10 independent test runs.

  - `pts/stream-1.3.4` for memory bandwidth

- **Workload Parameters:** Each test was executed with 10 warm-up runs followed by 50 measurement iterations. Results report mean values with 95% confidence intervals. For container workloads, we used a baseline nginx:alpine image and a compute-intensive Python image performing matrix operations.

- **Policy Configuration:** AppArmor profiles were set to enforce mode using Docker's default docker-default profile. eBPF-Secure policies were configured to match the same restriction level (blocking mount, rawio, and privileged syscalls).

All measurements were collected with system idle background processes minimized, and tests were conducted during off-peak hours to reduce external interference. The normalized overhead values reported in Table 4 are calculated as the ratio of confined execution time to unconfined baseline execution time[3, 8].

7. Discussion, Limitations, and Future Work

While `eBPF-Secure` presents a flexible and powerful framework for Linux container confinement, several considerations and limitations must be addressed to facilitate its widespread adoption and future development [12].

One of the key advantages of `eBPF-Secure` is its modular design. By decoupling enforcement logic from the kernel and allowing dynamic policy changes, it supports runtime adaptation to container behavior. This is a substantial improvement over static tools like AppArmor and seccomp. However, this flexibility comes at the cost of complexity. Writing secure and performant eBPF policies currently requires low-level knowledge of kernel structures, syscall arguments, and BPF map semantics[7].

To mitigate this, future work should focus on the development of high-level domain-specific languages (DSLs) or policy compilers. These tools could abstract low-level BPF primitives into readable security rules, enabling easier adoption by system administrators and DevOps practitioners. This would align eBPF policy management with established tools like SELinux's reference policy framework or AppArmor's profile generators.

Another challenge lies in kernel version compatibility. Because eBPF features (e.g., BPF links, LSM hook types) evolve rapidly across kernel versions, policies written for one environment may not be portable. Addressing this requires robust fallback mechanisms or version-aware compilers that gracefully degrade functionality without compromising security[12].

Scalability in large container orchestrators like Kubernetes is also an open concern. While `bpfcontain` uses per-cgroup maps for isolation, there's a need to coordinate these maps across thousands of pods. Integration with orchestration layers—via admission controllers or runtime shims—could provide hooks for applying and removing container-specific BPF policies at lifecycle events.

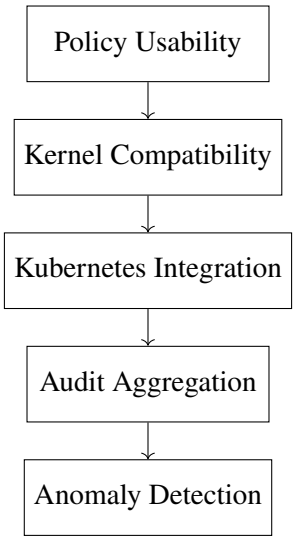

Figure 4: Future research and engineering priorities for eBPF-Secure organized by timeline and impact. Near-term priorities focus on usability enhancements (DSL development) and orchestration integration. Medium-term goals address scalability in large deployments and ML-driven anomaly detection. Long-term objectives target autonomous policy adaptation and formal verification of security properties. This roadmap reflects the transition from research prototype to production-ready security framework.

In addition to security, eBPF enables observability [1, 8] of `eBPF-Secure` can be expanded into automated response mechanisms. While current audit streams provide excellent visibility, linking these to alerting and runtime mitigation (e.g., container pause or restart) could enhance incident response workflows. This bridges the gap between passive monitoring and active defense.

Lastly, an exciting direction involves integrating machine learning (ML) models to identify abnormal system behavior. Since `eBPF-Secure` logs granular syscall data, these can be converted into feature vectors for online anomaly detection. Coupling this with reinforcement learning could enable policies that evolve in response to threats.

In conclusion, `eBPF-Secure` is a promising foundation for next-generation security and observability in containerized environments. Its modular, dynamic, and verifiable architecture addresses longstanding limitations of traditional confinement mechanisms. By addressing its current complexity and integrating with orchestration ecosystems, it can pave the way toward secure, scalable, and adaptive Linux container platforms[13].

8. Ethical and Security Considerations

The development and deployment of kernel-level instrumentation tools like eBPF-Secure carry significant responsibility. We acknowledge the following considerations:

8.1 Responsible Use Guidelines

eBPF programs execute within the kernel with elevated privileges. While the eBPF verifier ensures safety (preventing crashes or memory corruption), it does not guarantee that policies themselves are ethically sound or free from bias. Operators must[12]:

- Conduct thorough policy testing in staging environments before production deployment.
- Implement the principle of least privilege, granting only minimum necessary permissions.
- Maintain audit trails of policy changes for forensic analysis and compliance verification.

8.2 Potential for Misuse

Like any powerful security tool, eBPF-based frameworks could be misused for surveillance or unauthorized monitoring. eBPF-Secure mitigates this through:

- **Capability restrictions:** Loading eBPF programs requires `CAP_BPF` or root privileges, limiting attack surface.
- **Verifier enforcement:** Programs cannot bypass kernel security mechanisms or access arbitrary memory.
- **Transparency:** All loaded programs are visible via `bpftool` and kernel auditing interfaces.

8.3 Privacy Implications

The observability capabilities of eBPF-Secure collect detailed syscall traces. In multi-tenant environments, operators should:

- Anonymize sensitive data (e.g., file paths, network addresses) in audit logs.
- Establish clear data retention and deletion policies.
- Inform tenants about monitoring scope and purposes, complying with regulations like GDPR and HIPAA where applicable.

8.4 Recommendations for Safe Deployment

We recommend that organizations adopting eBPF-Secure:

1. Use signed eBPF programs with cryptographic verification where supported.
2. Implement defense-in-depth, combining eBPF enforcement with traditional LSMs.
3. Regularly update kernel versions to receive latest eBPF security patches.
4. Conduct red-team exercises to validate policy effectiveness.

These considerations align with broader industry efforts toward responsible AI and secure systems engineering, ensuring that innovation in container security does not compromise user trust or system integrity[1, 3].

References

[1] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge & container-based microservices," *IEEE Access*, vol. 10, pp. 86 904–86 919, 2022.

[2] A. Mahida, "Enhancing observability in distributed systems - a comprehensive review," *Journal of Mathematical & Computer Applications*, vol. 2, no. 135, pp. 2–4, 2023, dOI: 10.47363/Jmca/2023(2).

[3] U. Faseeha, H. J. Syed, F. Samad, S. Zehra, and H. Ahmed, "Observability in microservices: An in-depth exploration of frameworks, challenges, and deployment paradigms," *IEEE Access*, vol. 13, pp. 72 011–72 039, 2025.

[4] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, and T. Xu, "Programmable system call security with eBPF," 2023.

[5] S. Jero, J. Furgala, R. Pan, P. K. Gadepalli, A. Clifford, B. Ye, and R. Skowyra, "Practical principle of least privilege for secure embedded systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, May 2021, pp. 1–13.

[6] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Nov. 2022, pp. 1963–1976.

[7] L. Rice, *Learning eBPF*. O'Reilly Media, Inc., 2023.

[8] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: An industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, p. 25, 2022.

[9] C. Liu, B. Tak, and L. Wang, "Understanding performance of eBPF maps," in *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*. ACM, Aug. 2024, pp. 9–15.

[10] Y. Zhong, H. Li, Y. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, and A. Cidon, "XRP: In-Kernel storage functions with eBPF," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022, pp. 375–393.

[11] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, and Q. Li, "Cross container attacks: The bewildered eBPF on clouds," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023, pp. 5971–5988.

[12] S. Y. Lim, X. Han, and T. Pasquier, "Unleashing unprivileged eBPF potential with dynamic sandboxing," in *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*. ACM, Sep. 2023, pp. 42–48.

[13] S. Shankar and A. Parameswaran, "Towards observability for production machine learning pipelines," 2021.