Project Report: eBPF for Cloud-Native Security - From Kernel Observability to Runtime Enforcement

Wu Xiyu

No Institute Given

Abstract. The widespread adoption of cloud-native architectures, characterized by dynamic, ephemeral, and highly distributed microservices, has rendered traditional perimeter-based security models insufficient. This project explores the application of *extended Berkeley Packet Filter (eBPF)* as a foundational technology to address these new security challenges. eBPF allows for the safe execution of user-defined programs within the Linux kernel without modifying kernel source code or loading modules. This report investigates how eBPF enables deep system observability, fine-grained runtime security enforcement, and performance-efficient network security in containerized environments. We present a literature review, a conceptual system architecture for an eBPF-based security agent, an analysis of use cases and performance implications, a discussion of current limitations, and future research directions. Our findings indicate that eBPF represents a paradigm shift for cloud-native security, offering the granularity and context required to protect dynamic workloads without compromising system performance or stability.

Keywords: Cloud-Native Security · eBPF · Runtime Security · Observability · Linux Kernel.

1 Introduction

1.1 Motivation and Background

The shift to cloud-native computing, powered by containers, orchestrators like Kubernetes, and microservices, has introduced profound security complexities. The attack surface has expanded, boundaries have dissolved, and the life-cycle of workloads has accelerated [2]. Traditional security tools, often reliant on host-level agents and static rules, struggle with the transience and scale of modern deployments. There is a critical need for security solutions that are:

- **High-performance and Low-overhead:** To avoid impacting application SLOs.
- **Context-aware:** Understanding Kubernetes pods, namespaces, and labels.
- **Kernel-level:** To gain visibility and enforce policies beneath the application layer, where many attacks occur.

1.2 Project Contributions

This report makes the following contributions:

1. It synthesizes current research and practical implementations of eBPF for security, providing a clear taxonomy of its capabilities (Observability, Networking, Enforcement).
2. It proposes a high-level architecture for a cloud-native security runtime agent built on eBPF, detailing component interactions.
3. It analyzes key use cases—such as detecting privilege escalation and zero-day exploit attempts—and discusses the performance advantages over traditional tools like Auditd.
4. It outlines the current technical and ecosystem challenges that must be overcome for eBPF-based security to become ubiquitous.

2 Literature Review and Related Work

The concept of in-kernel programmability is not new. *Berkeley Packet Filter (BPF)* was originally designed for efficient network packet filtering [6]. Its extended successor, eBPF, has evolved into a general-purpose execution environment within the kernel [1].

eBPF for Observability. Tools like **BCC** and **bpftrace** [3] have popularized eBPF for dynamic tracing and performance analysis, providing unprecedented visibility into system calls, network traffic, and kernel functions. Projects like **Falco** [7], initially built on kernel modules, have migrated to eBPF as their default driver, highlighting its reliability and safety advantages for security monitoring.

eBPF for Networking and Security. The **Cilium** project [4] is a landmark implementation that uses eBPF to provide Kubernetes-aware network security, load balancing, and observability, replacing traditional iptables-based kube-proxy. This demonstrates eBPF's ability to enforce identity-aware (pod-level) network policies with superior performance.

Traditional Security Tools. Host-based Intrusion Detection Systems (HIDS) like **OSSEC** and Linux auditing frameworks (**Auditd**) rely on syscall auditing. These methods are often noisy, high-overhead, and lack the granular context of container orchestration platforms. eBPF offers a more efficient and targeted alternative by allowing programs to be attached to specific kernel tracepoints or probes.

3 System Architecture: An eBPF-Based Security Runtime Agent

We propose a conceptual architecture for a security agent that leverages eBPF as its core engine, designed to run as a DaemonSet in a Kubernetes cluster.

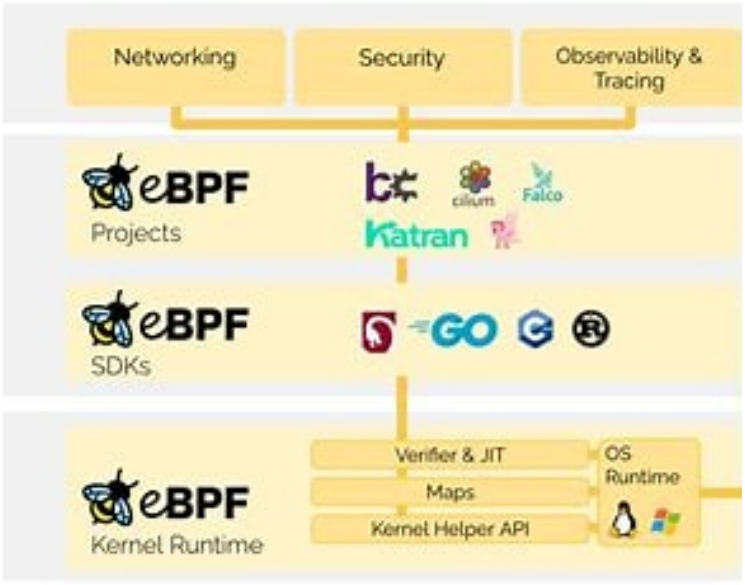

Fig. 1. High-level architecture of the eBPF-based security agent.

3.1 Components

1. **eBPF Program Suite:** A collection of small, purpose-built eBPF programs attached to critical kernel hooks:
   - `tracepoint/syscalls/sys_enter_execve` for process execution tracking.
   - `kprobe/commit_creds` for privilege escalation detection.
   - `tracepoint/sock/socket_connect` for network connection control.
   - `LSM (Linux Security Module) hooks` for file access and capability checks (via BPF-LSM).
2. **User-Space Agent (Go/Rust):** Manages the lifecycle of eBPF programs (load, attach, unload). It receives events from the eBPF programs via *perf rings* or *eBPF maps*, enriches them with Kubernetes metadata (e.g., Pod name, labels), and executes policy logic.

3. **Policy Engine:** A rule evaluation component that processes enriched events against a declarative security policy (e.g., YAML-based). Example rule: "`Block any process not from image 'myrepo/app:latest' from making outbound connections to port 22`".
4. **Control Plane Interface:** Provides an API for security administrators to deploy policies, query alerts, and manage the agent.
5. **Alerting and Logging:** Integrates with existing SIEM (e.g., Elasticsearch) and alerting (e.g., Prometheus Alertmanager) platforms.

3.2 Data Flow

1. An event (e.g., a system call) triggers an attached eBPF program in the kernel.
2. The eBPF program filters and captures relevant data (PID, UID, command, pathnames), writing it to a shared **eBPF map**.
3. The user-space agent polls the map, reads the event, and enriches it by querying the Kubernetes API to map the PID to a specific Pod and Namespace.
4. The enriched event is passed to the Policy Engine. If it violates a rule, an alert is generated and a mitigation action (e.g., killing the process via a signal sent back to the kernel via another eBPF map) may be triggered.

4 Use Cases and Performance Evaluation

4.1 Key Security Use Cases

1. Runtime Threat Detection. eBPF can detect attack patterns in real-time by tracing process lineage, file access, and network activity.

```
SEC("tracepoint/syscalls/sys_enter_fork")
int handle_fork(struct trace_event_raw_sys_enter *ctx) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    u64 *counter = fork_count.lookup(&pid);
    if (counter) {
        (*counter)++;
        if (*counter > FORK_THRESHOLD) {
            bpf_printk("Fork bomb detected in PID %d\\n", pid
    );
            // Trigger alert in user-space via map
        }
    }
    return 0;
}
```

Listing 1.1. Simplified eBPF code snippet to detect a fork bomb pattern.

2. Zero-Trust Network Security. Replacing iptables with eBPF allows for pod-identity-aware network policies. A policy can allow traffic from `app=frontend` to `app=backend` on port 8080, regardless of IP address changes, which is impossible with traditional firewall rules.

3. Vulnerability Exploit Mitigation. eBPF can hook into kernel functions targeted by exploits. For example, a program attached to `ptrace()` can block unexpected debugging attempts, potentially thwarting certain privilege escalation exploits.

4.2 Performance Comparison

A key advantage of eBPF is its performance. The following table summarizes a conceptual comparison based on published benchmarks [5]:

Table 1. Conceptual Performance and Capability Comparison

Criteria	**Auditd**	**Traditional HIDS**	**eBPF-Based Agent**
Overhead	Very High (audit backlog)	High	Very Low
Granularity	System-call level	File/Process level	Kernel event, network packet
Container Context	Limited (cgroup tracking)	Limited	Native (K8s labels, pod ID)
Enforcement Ability	Logging only	User-space actions	In-kernel actions possible

The efficiency stems from eBPF's just-in-time (JIT) compilation and its execution at the earliest possible point in the kernel's data path, avoiding costly context switches to user-space for filtering.

5 Limitations and Challenges

Despite its promise, eBPF faces several challenges for broad security adoption:

- **Kernel Version Dependency:** Advanced eBPF features (e.g., BPF-LSM, CO-RE) require recent kernels (5.8+ for full LSM support). This can be a barrier in enterprise environments.
- **Complexity:** Writing safe and correct eBPF programs requires deep kernel knowledge. While libraries (`libbpf`) and higher-level frameworks are improving, the barrier to entry remains high.
- **Stability of Hooks:** Kernel tracepoints are stable, but **kprobes** can be fragile, as they attach to function names that may change between kernel versions.
- **Evasion and Subterfuge:** A determined attacker with root privileges could potentially tamper with or disable eBPF programs. BPF-LSM and other integrity mechanisms aim to mitigate this.
- **Ecosystem Immaturity:** While projects like Cilium are mature, the broader ecosystem of eBPF security tools is still evolving, with gaps in centralized management and policy standardization.

6 Conclusion and Future Work

This project has explored eBPF as a transformative technology for cloud-native security. Its ability to provide deep, efficient, and context-aware observability and enforcement directly within the kernel aligns perfectly with the demands of dynamic container environments. It moves security closer to the workload, enabling a true zero-trust implementation at the kernel level.

6.1 Summary of Key Points

- eBPF enables high-performance, kernel-native security primitives for observability, networking, and runtime enforcement.
- It provides superior context for containerized workloads compared to traditional tools.
- An architecture combining eBPF programs with a user-space policy agent is the prevailing model for building security solutions.
- Performance benefits are significant, but adoption is tempered by kernel requirements and implementation complexity.

6.2 Recommendations for Further Research

1. **Policy as Code:** Developing standardized, declarative policy languages (beyond Rego or YAML) specifically designed for eBPF-powered security controls.
2. **Formal Verification:** Research into formally verifying the safety and correctness of eBPF security programs to prevent kernel crashes or security bypasses.
3. **AI/ML Integration:** Exploring how eBPF's rich data stream can feed real-time machine learning models for anomaly detection of never-before-seen attack patterns.
4. **Hardware Integration:** Investigating the intersection of eBPF and hardware offloading (e.g., SmartNICs) to push security enforcement even closer to the hardware for ultimate performance.

The journey of eBPF in security is just beginning. As the kernel and tooling mature, it is poised to become the default substrate for securing the next generation of cloud infrastructure.

References

1. eBPF.io: ebpf - introduction, tutorials & community resources. `https://ebpf.io`, accessed: 2025-12-19
2. Foundation, C.N.C.: Cncf cloud native interactive landscape. `https://landscape.cncf.io`, accessed: 2025-12-19
3. Gregg, B., et al.: Bpf compiler collection (bcc) and bpftrace. `https://github.com/iovisor/bcc`, `https://github.com/iovisor/bpftrace`, accessed: 2025-12-19

4. Isovalent: Cilium - ebpf-based networking, security, and observability. https://cilium.io, accessed: 2025-12-19
5. Isovalent: Cilium performance benchmarking. https://cilium.io/blog/2021/05/11/cilium-110#performance, accessed: 2025-12-19
6. McCanne, S., Jacobson, V.: The BSD packet filter: A new architecture for user-level packet capture **1993**, 259–270 (1993)
7. Project, T.F.: Falco: Cloud native runtime security. https://falco.org, accessed: 2025-12-19

A Project Code Repository

The code developed for this conceptual analysis, including example eBPF programs and a simple user-space loader, is available in the GitHub repository: https://github.com/yudazhuang666/ebpf-security-project.

Note: The repository contains only self-written code and adaptations. It does not include full source code for external dependencies like libbpf or the Linux kernel headers.