

Differential Observability: Detecting Silent State Divergence in eBPF-based Cloud Networks

Ankita Ojha
Google

Abstract

In modern Kubernetes environments, eBPF (Extended Berkeley Packet Filter) has become the de facto standard for high-performance dataplane enforcement. However, this architecture introduces a complex distributed state problem: the asynchronous interplay between the Kubernetes control plane (intent) and the kernel-space BPF maps (reality).

A critical failure mode, termed “Silent Divergence,” occurs when the control plane believes a network policy or identity is applied, but the underlying kernel state is missing or corrupted. In this “Gray Failure” state, standard observability tools—including logs, liveness probes, and agent status checks—report health, while the network silently drops traffic.

This paper introduces eBPF-Auditor, a specialized consistency verification framework. Unlike agents that rely on event-based reconciliation, eBPF-Auditor performs a periodic “Two-Way State Audit” that mathematically verifies the intersection of Kubernetes Intent and eBPF Reality. We demonstrate through fault injection and benchmarks on 5,000 Pods that this approach successfully detects state drift with 100% accuracy and negligible sub-millisecond overhead (ms), making it a viable solution for high-frequency runtime verification in production hyperscale Kubernetes clusters.

Keywords: eBPF, Kubernetes, Network Reliability, Gray Failures, Distributed Systems, Cloud Computing, Silent Divergence

Introduction

eBPF-based networking (e.g., Cilium, Calico eBPF) in Kubernetes has fundamentally decoupled the “intent” of the network from its “reality”. The intent is managed by the Kubernetes API Server, representing the desired state of Pods, Services, and Policies. The enforcement is handled by the Linux kernel, specifically within maps that store forwarding tables and security identities.

While efficient, this separation relies on a local user space agent to synchronize thousands of state updates per minute. We identify a dangerous failure mode in hyperscale clusters: Silent Divergence. This phenomenon occurs when a state update is lost due to a race condition, hardware offload failure, or manual intervention, leaving the kernel map inconsistent with the agent's internal memory. Because the agent's internal cache reflects "success," it never retries, and because it sees no errors from the kernel, it logs nothing.

The Challenge of State Reconciliation

Traditional reconciliation loops in Kubernetes are "event-triggered" but often rely on the assumption that the underlying actuator (the agent) is infallible once it acknowledges an event. However, in eBPF systems, the "database" is raw kernel memory, which can be altered by external factors (e.g., bpftool, other programs, or kernel bugs) without notifying the userspace agent. This creates a blind spot where the system is ostensibly healthy but functionally broken.

Contributions

This paper validates the need for an independent "two-way state audit" pattern. We make the following contributions:

- **Failure Taxonomy:** We define and classify "Silent Divergence" scenarios where event-driven agents fail to detect differences between the intent and reality.
- **The Auditor Algorithm:** We present a formal logic for cross-referencing intent ($\mathbb{I}$) and Reality ($\mathbb{R}$) using a differential set approach.
- **Empirical Evaluation:** We provide empirical proof using fault injection to simulate invisible network failures and demonstrate the auditor's detection capabilities with minimal CPU overhead.

Related Work

The challenge of observing distributed systems has evolved significantly with the introduction of kernel-bypass and eBPF technologies.

Kubernetes Observability

Standard Kubernetes observability relies on Liveness and Readiness probes. These operate at Layer 7 (HTTP) or Layer 3 (TCP Socket) but are initiated from user space. While effective for application crashes, they often produce "false positives" in networking scenarios. If a node's local BPF map drops packets, the probe may timeout, but the Kubelet attributes this to application load rather than dataplane problem.

eBPF-based Monitoring

Tools like Hubble and Tetragon provide deep visibility into kernel flows. However, they largely rely on the same datapath constructs as the enforcement agent. If a map entry is missing, the traffic is often dropped before the monitoring hook can attribute it to a specific Pod, leading to unidentified drops that are difficult to correlate with an application problem.

Data Plane Verification in SDN

Our work draws inspiration from Data Plane Verification in Software Defined Networking (SDN). Tools like Veriflow and Hassel verify network-wide constants by checking the forwarding tables of physical switches against the controller's intent. eBPF-Auditor adapts this concept to the host-local scope of Kubernetes nodes, treating the Linux kernel as a programmable switch that requires independent verification.

Gray Failures

Huang et al. defined “Gray Failures” as differential observability states where a system component believes it is healthy while failing. Our work extends this definition to the specific domain of kernel configuration in container orchestration systems.

System Design: The Two-Way Audit

To solve the silent divergence problem, we developed eBPF-Auditor, a lightweight consistency engine. Unlike the other agent based approaches, which optimizes for write-performance, eBPF-Auditor optimizes for correctness by performing a two-way state audit periodically.

Architecture

eBPF-Auditor runs as a DaemonSet on every node. It operates in a continuous loop with a configurable interval (default: 30s). It requires the CAP_SYS_ADMIN capability to read BPF maps via the `bpf()` syscall.

Phase 1: Intent Construction ($\mathbb{I}$)

eBPF-Auditor queries the local Kubelet API or watches the Kubernetes API Server to build a deterministic set of expected active endpoints. To minimize load on the API server, we utilize a local informer cache. In this step, eBPF-Auditor determines the Kubernetes control plane intent.

Phase 2: Reality Construction ($\mathbb{R}$)

eBPF-Auditor dumps the raw memory of the cilium_ipcache BPF map. We utilize `bpftool` (or a Go library) to iterate over the map elements. In this step, eBPF-Auditor parses the hexadecimal keys and values into a set of programmed entries in order to determine the programmed reality. eBPF-Auditor performs this step without relying on any CNI plugin.

Phase 3: Differential Logic

The core innovation is the bidirectional set difference check:

- *Input:* $\mathbb{I}$ (K8s Pods), $\mathbb{R}$ (BPF Map)
- *Output:* Divergence Report
- $\mathbb{D}_{\text{silent}} = \mathbb{I} \setminus \mathbb{R}$
- $\mathbb{D}_{\text{leak}} = \mathbb{R} \setminus \mathbb{I}$
- If $\mathbb{D}_{\text{silent}} \neq \emptyset$, **Alert:** “Silent Divergence Detected”
- If $\mathbb{D}_{\text{leak}} \neq \emptyset$, **Warn:** “Ghost Entries Detected”

This logic detects two distinct failure modes that are classified as Silent Divergence:

- ($\mathbb{D}_{\text{leak}}$): IPs that exist in the kernel but not in K8s (e.g., Leaked IPs from deleted pods).
- ($\mathbb{D}_{\text{silent}}$): The critical “Blackhole” scenario where active Pods are missing from the dataplane.

Evaluation

We validated eBPF-Auditor against the “Silent Divergence” scenario on a live GKE cluster running Dataplane V2.

Experimental Setup

We utilized a cluster with standard eBPF networking enabled.

- **Victim Pod:** rtt-client (IP: 10.36.0.22).
- **Traffic Generator:** Continuous TCP ping to measure connectivity.
- **Environment:** Linux Kernel 5.15+, Cilium-based CNI.

Fault Injection

We simulated a silent divergence by manually modifying kernel memory, bypassing the agent. This represents the class of failures where the Kubernetes control plane is unaware of data plane state changes.

Example simulating silent divergence by removing an IP address from the kernel map:

```
None
BASH
kubect1 exec -n kube-system anetd-node -- \
```

```
bpftool map delete pinned \  
/sys/fs/bpf/tc/globals/cilium_ipcache \  
key 0x0a 0x24 0x00 0x16 ...
```

Results

The impact was immediate, but invisible to standard tools.

The Observability Gap

Standard observability tools failed to detect the drift (Table 1):

- Traffic: curl requests timed out (100% loss).
- Agent Logs: Zero error messages produced.
- Health Status: Agent reported “OK”.

Table 1: The Observability Gap During Failure

Tool	Status Reported or Symptoms Observed	Actual State
K8s API	Running / Ready	Broken ▾
Cilium Agent	Healthy / Synced	Stale ▾
Agent Logs	No Errors logged	Silent ▾
Data Plane	Packet Drop / Timeout	Broken ▾
eBPF-Auditor	[SILENT_DIVERGENCE]	Detected ▾

Fault Injection & Recovery Analysis

We performed a live reproduction of the "Silent Divergence" failure mode on the production cluster.

- **Injection:** We surgically removed the ipcache entry for a valid source Pod IP (10.36.6.11) on a destination node using bpftool.
- **Impact:** Connectivity immediately failed. The application reported `CURLE_OPERATION_TIMEDOUT` (Exit Code 28) as the traffic was dropped by the default deny policy due to missing identity.
- **Silence:** Throughout this outage, the resident anetd agent logs showed zero errors or warnings.
- **Recovery:** Upon forcibly restarting the anetd Pod, connectivity was restored within 2 seconds. The new agent instance "re-populated" the map during its initialization phase without flagging the pre-existing discrepancy. This confirms that without a tool like eBPF auditor, such divergences persist until a random or scheduled restart event occurs.

Live Verification

We deployed eBPF-Auditor on a live GKE cluster to verify its non-disruptive nature. In a healthy state, the Auditor correctly identified 0 "Silent Divergences". However, it flagged 5 "Ghost Entries" corresponding to infrastructure IPs (e.g., Node IP, Gateway, Link-Local), demonstrating its ability to surface all kernel state, which can then be allowlisted or audited for unauthorized backdoors.

Performance Benchmark

To verify the feasibility of running this audit in production, we benchmarked the differential logic against large-scale state. We simulated a node hosting 5,000 active Pods and measured the execution time of the set difference operation ($I \setminus R$).

Table 2: eBPF-Auditor Performance (N=5000 Pods, 100 Iterations)

Metric	Value
Average Processing Time	0.1866 ms
P99 Processing Time	0.2835 ms
Accuracy	100%

CPU Overhead	Negligible ()
--------------	---------------

As shown in Table 2, the core logic executes in less than 0.2 milliseconds. Even including the overhead of *bpftool dump* (approx. 50-100ms for large maps), the total audit cost is minimal, allowing for safe execution intervals of 10 seconds without impacting node performance.

Root Causes of Divergence

Our research identifies three primary technical drivers for the state drift detected by eBPF-Auditor.

LRU Map Eviction & Saturation

High-performance eBPF dataplane agents often utilize BPF_MAP_TYPE_LRU_HASH for state tables (e.g., Connection Tracking or NAT) to prevent memory exhaustion. By design, these maps evict the Least Recently Used (LRU) entries when full. However, in hyperscale environments with high connection churn, valid active entries can be evicted if the map sizing is insufficient (e.g., default 64k entries).

Crucially, our analysis of the anetd logs during these failure scenarios revealed zero error messages. This confirms that the Linux kernel does not emit a userspace signal (e.g., via netlink or epoll) when an LRU eviction occurs. The agent continues to believe a policy is active, while the kernel silently drops packets associated with the evicted flow. We believe this is the most common cause of "Silent Divergence" in production.

Asynchronous Agent Restarts

Silent divergence can occur during a rolling update or crash recovery of the node agent. The new agent instance must reconstruct the desired state from the Kubernetes API and reconcile it with the existing BPF maps. If this reconciliation logic is non-atomic (e.g., clearing a map in one step before repopulating it in another step), there is a period of time where traffic could be dropped. Furthermore, if the agent crashes during an update, the BPF map might be partially updated and inconsistent with both the old and new intents.

Atomic Update Failures

While individual BPF map updates are atomic, complex state changes often require multiple syscalls (e.g., updating a Policy Map and then an Endpoint Map). There is no "transaction" concept across multiple BPF syscalls. If an error occurs halfway through this sequence (e.g.,

ENOMEM), the system enters an undefined state where some maps reflect the new intent while others reflect the old, leading to undefined forwarding behavior.

Discussion

Race Conditions

A potential limitation of the “Snapshot” approach is the consistency window. If a Pod is created at t_{start} , the agent updates the map at t_{map} . If eBPF-Auditor runs exactly in this $t_{\text{start}} - t_{\text{map}}$ window, it may flag a false positive. To mitigate this, we implement a grace period where divergences must persist for N consecutive audit cycles before eBPF-Auditor generates an alert.

Auto-Remediation

Beyond detection, eBPF-Auditor lays the groundwork for self-healing systems. Since eBPF-Auditor possesses the “Truth” ($\mathbb{I}$), it could theoretically reinject the missing keys into the BPF map. However, our current recommendation is to trigger a “Pod Restart” or “Agent Restart” signal, forcing a safe reconciliation path rather than directly modifying kernel memory, because directly modifying kernel memory carries a higher risk.

Conclusion

Our results demonstrate that eventual consistency is insufficient for high-reliability cluster networking. As eBPF pushes more logic into the kernel, the gap between user-space intent and kernel-space reality grows.

We propose that cloud providers adopt the pattern that we introduce with eBPF-Auditor: a periodic, read-only reconciliation loop. By treating the kernel map as an untrusted dataset that must be verified against Kubernetes, systems can self-heal from “Gray Failures” that currently require manual intervention.

References

- P. Huang et al., “Gray Failure: The Achilles’ Heel of Cloud-Scale Systems,” in Proc. HotOS, 2017.
- Cilium Project, “Cilium: eBPF-based Networking, Security, and Observability,” [Online]. Available: <https://cilium.io>.
- Kubernetes Authors, “Kubernetes Documentation,” [Online]. Available: <https://kubernetes.io>.
- Hubble: Cilium Project, “Hubble: eBPF-based Networking, Security, and Observability,” [Online]. Available: <https://github.com/cilium/hubble>.

- Tetragon: Cilium Project, “Tetragon: eBPF-based Security and Observability,” [Online]. Available: <https://github.com/cilium/tetragon>.
- Veriflow: A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, “VeriFlow: Verifying Network-Wide Invariants in Real Time,” in *Proc. USENIX NSDI*, 2013.
- Hassel (Header Space Analysis): P. Kazemian, G. Varghese, and N. McKeown, “Header Space Analysis: Static Checking for Networks,” in *Proc. USENIX NSDI*, 2012.