AWS LAMBDA USES eBPF TO SCALE NETWORK FLOW CAPTURE FROM 150K TO 4.7 MILLION CONNECTIONS PER HOST

Overview

AWS Lambda runs thousands of Firecracker microVMs per bare-metal host, each belonging to a different customer and sometimes living only a few hundred milliseconds. Lambda needs a complete, correctly attributed network flow record for every one, for security, audit, compliance, and metering. The legacy capture system relied on iptables rules and a kernel module inherited from a lower-density, single-tenant design, and could not scale to Lambda's density or see IPv6 traffic. Lambda replaced it with an observe-only eBPF capture layer and unprivileged Rust userspace aggregators, removing the per-packet rule-walk tax, adding full IPv6 coverage, and producing byte-identical output so no downstream consumer had to change.

4.7 million

connections handled in a burst from one production workload

Challenge

Lambda needed to record every network flow on densely packed multi-tenant hosts without slowing traffic or the platform:

- **Attribution:** every packet and flow must map to the correct microVM and tenant, since logs feed billing, compliance, and incident response.
- **Rule explosion:** a host running ~2,000 microVMs needed 100,000+ iptables rules, with cost growing as the host filled.
- No IPv6 support in the legacy kernel module.
- **Near-zero overhead:** CPU and memory costs multiply across thousands of microVMs per host.
- **No breaking changes:** dozens of internal systems consume the existing Ion records.

Solution

Lambda built a three-part pipeline on each worker host:

- **eBPF capture layer:** observe-only programs on tc (clsact) ingress/egress hooks per microVM network, parsing Ethernet, IPv4/IPv6, and TCP/UDP/ICMP headers into a compact event (~24 bytes for IPv4) per packet, written to a dedicated BPF ring buffer.
- **Rust tagger:** one unprivileged userspace process per microVM network, draining its ring buffer into per-flow Ion records.
- **Orchestrator:** one privileged process per host, with only two capabilities, that loads eBPF programs, configures tc, supervises taggers, and exposes a gRPC lifecycle API.

The design isolates tenants with one ring buffer and device each, passes ring buffer file descriptors via SCM_RIGHTS so taggers need no open permissions, sizes buffers from real packet rates, and wakes userspace only when a ring is ~1% full, draining at most every 100 ms, so thousands of processes never thrash the CPU.

Results

The move to eBPF delivered:

- **No more 150,000-connection ceiling:** the system has handled a burst of 4.7 million connections from one production workload.
- IPv6 flow logs already paying off, helping narrow a network instability issue to a downstream service.
- Elimination of 100,000+ per-host iptables rules, replaced by constant-time map lookups that don't scale with host density.
- A few hundred KB of RAM per tagger against a ~1 MB budget.
- Activation latency under 2 ms at p90 and under 10 ms at p99.9, matching the legacy system.
- Byte-for-byte identical output, enabling zero downstream migration.
- **A smaller privileged surface:** none of the processes touching customer traffic hold privileges.

Future Plans

Lambda is building further improvements with Arenas and classifiers to modernize its networking stack off iptables, and to shrink the memory footprint of its ring buffers.

Why eBPF?

eBPF let Lambda watch traffic from outside the forwarding path instead of inline:

- Low, flat per-packet overhead, independent of how many microVMs share the host.
- **Verifier-proven safety:** programs are memory-safe, terminating, and observe-only by construction.
- **Flexible attach points:** tc hooks see every packet early, on both ingress and egress.
- **Efficient transport:** BPF ring buffers use reserve-then-commit with no syscall copy.
- **No kernel module to maintain:** new protocol support is a program change.
- Full source auditability in the kernel tree, with built-in regression testing.
- A forward-compatible path to future kernel features, such as Arena support.
- An upstream contribution path that keeps the implementation aligned with mainline.

Implementation Process

Moving from prototype to production surfaced issues at every layer, each resolved with a targeted fix:

- **Verifier limits:** the parser became a shared, once-verified subroutine, with the IPv6 extension-header walk bounded to a fixed hop count.
- **Edge cases:** non-first IPv4 fragments report zero ports/flags instead of garbage; GSO/GRO byte counts are handled correctly.
- **Correctness:** every build runs the C eBPF programs through the CBMC formal model checker.
- **Lock contention:** attaching to devices takes a host-wide lock, so the API was split into a heavy batched Create call and a lightweight Activate call.
- **Process churn:** taggers can now be reused for workloads that recycle networks, instead of always respawning.

How This Differs From Other Solutions

The prior closed-source C kernel module was poorly documented, poorly understood, and had stagnated; a new bespoke module or DPDK both carried heavy maintenance overhead. eBPF differs from the alternatives:

- **vs. iptables/netfilter:** cost doesn't scale with rule count or density.
- **vs. custom kernel modules:** verifier-checked and observe-only, not privileged code.
- **vs. pcap/sidecars:** only compact metadata leaves the kernel, no packet copies.
- **vs. sampled flow logs:** every packet is counted, as audit and metering require.
- **Isolation by construction:** per-tenant rings mean no after-the-fact attribution.

Closing Lessons for the eBPF Community

- Observe from outside the hot path.
- Size buffers from real limits.
- Isolate tenants at capture.
- Pass file descriptors to keep workers unprivileged.
- Keep output formats stable when swapping engines.