Advanced Architectures in Linux Runtime Security: A Comprehensive Analysis of eBPF-Based Telemetry and Enforcement Systems

1. Introduction: The Paradigm Shift in Kernel Observability

The landscape of Linux runtime security has undergone a radical transformation over the last decade, moving from reactive, signature-based detection to proactive, behavioral enforcement. Historically, security tools relied on kernel modules (LKMs) or slow, race-condition-prone APIs like ptrace and auditd. While LKMs offered deep visibility, they introduced significant stability risks; a bug in a kernel module could panic the entire system, a risk unacceptable in modern mission-critical production environments. Conversely, auditd provided stability but often suffered from performance bottlenecks under high load and lacked the granular context required for containerized environments.

The emergence of the Extended Berkeley Packet Filter (eBPF) has fundamentally altered this calculus. Originally designed for network packet filtering, eBPF has evolved into a general-purpose execution engine within the Linux kernel. It allows developers to run sandboxed, verified programs in kernel space without modifying the kernel source code or loading vulnerable modules. This capability has birthed a new generation of security tools—Falco, Tracee, and Tetragon—that promise the performance of kernel-native execution with the safety of a sandbox.

This report provides a rigorous, expert-level technical analysis of these three industry-standard tools. It dissects their internal architectures, specific kernel hooking strategies, event transport mechanisms, and rule implementation engines. Furthermore, drawing upon the lessons learned from these existing solutions, this document delineates the architectural design for **AegisBPF**, a theoretical, portfolio-grade runtime security agent implemented in C++ and eBPF. This design distinguishes between "core" functionality—essential for high-fidelity monitoring—and "overkill" features that introduce unnecessary complexity, addressing specific engineering challenges such as process ancestry tracking, PID recycling, and variable-length data handling in ring buffers.

1.1 The Theoretical Foundation: The eBPF Security Primitive

To analyze the architecture of these tools, one must first understand the specific eBPF primitives they leverage. Security observability relies on the ability to intercept kernel

execution paths.

The Verification Gauntlet
Unlike kernel modules, eBPF programs must pass a rigorous verification process before loading. The verifier ensures the program terminates (preventing infinite loops), does not access invalid memory (preventing crashes), and adheres to complexity limits (originally 4096 instructions, now significantly higher). This constraint dictates the architecture of all eBPF security tools: complex logic, large state tables, and string parsing must often be offloaded to userspace or handled via specific, bounded helpers.1
Hook Points: The Eyes of the System

- **Tracepoints:** Static markers embedded in the kernel source code (e.g., sys_enter_execve). They provide a stable Application Binary Interface (ABI) but are limited to the arguments exposed by the developer at compile time.
- **Kprobes (Kernel Probes):** Dynamic instrumentation that can attach to virtually any kernel function (e.g., do_execve). They offer visibility into internal kernel structures but are unstable across kernel versions, as internal function signatures change.
- **LSM (Linux Security Module) Hooks:** BPF-LSM allows eBPF programs to attach to the same authorization hooks used by SELinux and AppArmor. This is the mechanism for *enforcement*, allowing operations to be blocked before execution.[3]

2. Deep Technical Analysis of Existing Ecosystem

2.1 Falco: The Alerting Powerhouse

Falco, originally created by Sysdig and now a CNCF project, is the most established tool in this domain. Its architecture is characterized by a "kernel-heavy capture, userspace-heavy analysis" design philosophy.

2.1.1 Internal Architecture and The Driver Model

Falco's architecture is bifurcated into the **driver** (kernel space) and the **userspace engine**. This separation is managed via libscap (System CApture Library) and libsinsp (System INSPection library).

The Driver Layer
Falco supports multiple driver backends to maximize compatibility:

1. **Kernel Module (falco-probe):** The legacy implementation. It inserts a .ko module that hooks system calls via tracepoints. It allocates a shared ring buffer (per CPU) to push events to userspace.
2. **Modern eBPF Probe:** This is the current standard for modern kernels (v5.8+). It is a CO-RE (Compile Once, Run Everywhere) enabled application. Unlike the legacy eBPF probe which required compiling the BPF bytecode on the target machine (to match kernel headers), the modern probe uses BTF (BPF Type Format) to adapt memory offsets

dynamically at runtime.[4]

Event Flow and Transport
The core event flow in Falco is syscall-centric:

1. **Capture:** A syscall (e.g., openat) triggers the eBPF program attached to the raw_tracepoint/sys_enter hook.
2. **Context Retrieval:** The BPF program retrieves the syscall arguments. For pointer arguments (like filenames), it uses bpf_probe_read_str.
3. **Serialization:** The event is serialized into a struct ppm_evt_hdr followed by the payload. This structure includes the timestamp, thread ID (TID), and event type.[5]
4. **Transport:** The data is pushed to a shared buffer. The modern probe utilizes the BPF_MAP_TYPE_RINGBUF, a multi-producer, single-consumer ring buffer that solves memory ordering issues inherent in the older BPF_MAP_TYPE_PERF_EVENT_ARRAY used by the legacy probe and module.[6]
5. **Userspace Consumption:** libscap polls the ring buffer.
6. **State Reconstruction:** libsinsp consumes the raw stream. It maintains a "thread table" in userspace memory, reconstructing the process tree, file descriptors, and container context (by querying the container runtime socket).[5]

2.1.2 Rule Implementation and Engine

Falco's rule engine operates almost entirely in userspace. Rules are defined in YAML and compiled into a filtering tree.

- **Evaluation:** When libsinsp produces an enriched event, it traverses the rule tree.
- **Syntax:** Rules rely on macros and lists (e.g., evt.type = open and fd.filename startswith /etc).
- **Kernel-Side Filtering:** Recent versions have introduced limited kernel-side filtering to drop high-volume, low-interest events (like read/write on unrelated file descriptors) to reduce the serialization overhead. However, the complex logic remains in userspace.[4]

2.1.3 Limitations

The primary architectural limitation of Falco is the TOCTOU (Time-of-Check-Time-of-Use) gap. Because the enforcement (killing a process) happens in userspace after the event has been serialized and analyzed, the malicious operation in the kernel has likely already completed. Furthermore, the serialization of syscall arguments for userspace analysis incurs a CPU penalty, typically 5-10% in high-throughput environments.[7]

2.2 Tracee: The Forensics Specialist

Tracee, developed by Aqua Security, differentiates itself by focusing on deep forensics and artifact capture, leveraging a Go-based userspace architecture.

2.2.1 Pipeline Architecture

Tracee relies on libbpfgo, a Go wrapper around the upstream libbpf C library. This allows it to seamlessly integrate eBPF management with Go's rich concurrency primitives.

- **Tracee-eBPF:** The kernel collector. It uses a mix of tracepoints and kprobes.
- **Tracee-Rules:** The detection engine.

2.2.2 Advanced Hooking and Artifact Capture

Tracee goes beyond metadata to capture payloads. This is achieved through specific hooks:

- **security_file_open (LSM) / vfs_write (Kprobe):** Tracee hooks these functions to intercept file writes.
- **Artifact Capture:** When a "suspicious" file write is detected (based on policy), Tracee doesn't just log the event. It captures the data buffer being written to disk and sends it to userspace. This allows security teams to extract dropped malware binaries for reverse engineering.[9]
- **Memory Artifacts:** It can capture memory regions of executing processes, useful for detecting fileless malware (e.g., code injection into running processes).[10]

2.2.3 Event Flow and PID Recycling

Tracee implements a rigorous pipeline for event ordering. Since events from different CPUs can arrive out of order in a perf buffer, Tracee-eBPF performs sorting in userspace to ensure causal consistency.
PID Recycling: Tracee addresses the issue of PID reuse (where the kernel reuses a PID after a process dies) by tracking the start_time of the process. When an event arrives, Tracee compares the start_time in the event metadata with its cached process table. If they differ, it knows the PID has been recycled and flushes the old process context.11

2.3 Tetragon: The Enforcement Enforcer

Tetragon, by Isovalent (creators of Cilium), represents the "enforcement-first" generation. It moves the decision logic *into* the kernel to enable synchronous blocking.

2.3.1 Internal Architecture: The Smart Sensor

Tetragon's architecture minimizes data movement. Instead of sending all events to userspace for evaluation, it loads "TracingPolicies" which are compiled into eBPF maps and logic.

- **Selectors:** Tetragon defines in-kernel filters called Selectors. A selector might specify: "On fd_install, if the filename is /etc/shadow AND the namespace is frontend, take action."
- **State Mapping:** Tetragon maintains significant state in kernel maps, including socket mappings and process ancestry, allowing it to make complex decisions without userspace round-trips.[13]

2.3.2 Enforcement Mechanisms

Tetragon employs two primary primitives for enforcement, effectively closing the TOCTOU gap:

1. **bpf_send_signal:** This helper allows the eBPF program to send a signal (typically SIGKILL) to the current process. This happens *immediately* when the hook triggers. If a process attempts a forbidden execve, it is killed before the new binary code executes.[15]
2. **bpf_override_return:** (x86 only). This allows the BPF program to hijack the return value of a kernel function. If a process calls open(), Tetragon can inject an -EPERM (Operation Not Permitted) error. The syscall fails gracefully (from the kernel's perspective), and the application receives an error code instead of a file handle.[13]

2.3.3 Identity and Ancestry: exec_id

Tetragon solves the PID recycling and namespace collision problem by synthesizing a cluster-unique identifier called exec_id.

- **Composition:** exec_id is a composite key derived from the 64-bit nanosecond boot time, the PID, and the node identifier.
- **Usage:** This ID is attached to every event. Even if PIDs are reused or overlap across containers, the exec_id remains unique for the lifetime of that specific execution instance, enabling precise lineage tracking.[17]

2.4 Comparative Summary

Feature	Falco	Tracee	Tetragon
Primary Focus	Threat Detection & Alerting	Forensics & Artifact Capture	Enforcement & Prevention
Architecture	Kernel-heavy capture, Userspace logic	Go pipeline, Artifact extraction	In-Kernel logic & filtering
Rule Engine	Userspace (Falco Rules)	Userspace (Rego/Go)	Kernel (TracingPolicy/CRD)
Enforcement	Async (Reactive, via sidekick)	Async (Reactive)	Sync (Inline, via Signal/Override)
Transport	Ring Buffer / Perf Buffer	Perf Buffer (Sorted in userspace)	Ring Buffer

Identity	PIDs & Container Socket	PIDs + Start Time	Unique exec_id

3. AegisBPF: Designing a Portfolio-Grade Security Agent

Based on the analysis of the ecosystem, we propose the design for **AegisBPF**, a high-performance, portfolio-grade runtime security agent. The design philosophy centers on **Efficiency**, **Resilience**, and **Safety**. It rejects the heavy userspace processing of Falco and the complexity of Go runtimes (garbage collection pauses) in favor of a lean C++ architecture.

3.1 Architectural Philosophy: Overkill vs. Core

To build a professional-grade agent, one must distinguish between essential capabilities and "bloat."

Core Requirements (Must-Have):

- **Process Ancestry:** A security event is meaningless without context. Knowing *who* spawned the malicious process is as important as the process itself.
- **Container Awareness:** Mapping PIDs to Container IDs (cgroups) is non-negotiable for cloud-native security.
- **File & Network Integrity:** Monitoring critical syscalls (openat, execve, connect, accept, bind, mprotect).
- **Signal-Based Enforcement:** The ability to synchronously kill processes violating policy is the defining feature of modern agents.[15]
- **Resilient Identity:** Handling PID recycling and namespaces correctly.

Overkill (Avoid):

- **L7 Protocol Parsing in Kernel:** Attempting to parse HTTP bodies or DNS packets entirely in eBPF is error-prone, computationally expensive, and hits verifier limits. This should be offloaded to userspace or dedicated proxies.[19]
- **Full System Call Tracing:** Tracing all 300+ syscalls generates massive noise. A focused set of ~20 security-critical syscalls covers 99% of threat vectors.
- **Complex Regex in Kernel:** Implementing full regex engines in BPF instructions is inefficient. Use prefix/suffix matching or Bloom filters for rapid rejection.[17]

3.2 Technology Stack

- **Userspace Language: C++20**. C++ provides RAII (Resource Acquisition Is Initialization) for robust resource management (automatically closing file descriptors and maps), zero-overhead abstractions, and seamless integration with libbpf. Unlike Go, it avoids

garbage collection pauses which can cause event drops in high-throughput scenarios.[20]

- **eBPF Library: libbpf** (upstream). We strictly use CO-RE. This removes the dependency on Clang/LLVM at runtime and ensures the agent runs on any BTF-enabled kernel without recompilation.[22]
- **Event Transport: BPF Ring Buffer**. We prioritize BPF_MAP_TYPE_RINGBUF over perf buffers for its strict ordering guarantees and ability to handle variable-length data more efficiently (via bpf_ringbuf_reserve).[6]

3.3 Kernel-Space Architecture and Map Design

3.3.1 Process Ancestry and PID Recycling

The heart of AegisBPF is the process_tree map. This map acts as a shadow process table in the kernel.

Map Structure:

C

```
struct process_key {
    u32 pid;
};

struct process_info {
    u32 pid;
    u32 ppid;
    u32 tgid;
    u64 start_time;      // Critical for PID recycling detection
    u64 parent_start_time;
    u32 uid;
    u32 gid;
    char comm;
    u32 container_id_hash; // Linkage to container metadata
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 32768);
    __type(key, struct process_key);
    __type(value, struct process_info);
} process_tree SEC(".maps");
```

Handling PID Recycling:
The kernel recycles PIDs. A naive map using only PID as a key will eventually map a new process to an old, dead ancestor. AegisBPF solves this using the start_time invariant.

1. **Capture:** On sched_process_fork or sys_enter_execve, the agent retrieves the current task's start time using BPF_CORE_READ(task, start_time) and the current time using bpf_ktime_get_ns().
2. **Verification:** When looking up a parent in the map (to link ancestry), the agent compares the start_time recorded in the map with the real_parent->start_time of the current task.
3. **Resolution:** If the start times do not match, the PID has been recycled. The agent treats the parent as "unknown" or orphaned rather than linking it to the wrong ancestor, preventing false lineage trails.[12]

3.3.2 The Enforcement Layer: Bloom Filters

Checking every file access against a massive denylist (e.g., 10,000 bad file paths) in the kernel is slow using standard Hash Maps due to string hashing overhead. AegisBPF utilizes a **Bloom Filter Map** (BPF_MAP_TYPE_BLOOM_FILTER) for the first line of defense.

Mechanism:

1. **Populate:** Userspace populates the Bloom Filter with hashes of all forbidden filenames.
2. **Fast Check:** On sys_enter_openat, the BPF program hashes the target filename.
3. **Peek:** It calls bpf_map_peek_elem on the Bloom Filter.
    - **Result 0 (Not Found):** The file is definitely safe (relative to the denylist). The syscall proceeds immediately. Cost: O(1).
    - Result 1 (Maybe Found): The file might be malicious. The BPF program then performs a lookup in a secondary, precise Hash Map to confirm. If confirmed, bpf_send_signal(SIGKILL) is invoked.
      This "Check-Reject-Verify" pattern minimizes the performance impact on legitimate operations, a critical requirement for production systems.25

3.3.3 Handling Variable Length Data

Sending variable-length data (like file paths up to PATH_MAX = 4096 bytes) over a ring buffer is tricky because bpf_ringbuf_reserve requires a compile-time constant size in older verifiers, or a runtime size that fits within strict limits.

AegisBPF Strategy:

1. **Header Reservation:** We reserve a fixed-size header struct in the ring buffer.
   C
   ```
   struct event_hdr {
     u32 event_type;
     u32 pid;
     u32 filename_len;
   ```

```
};
```

2. **Dynamic Pointer (Dynptr):** On kernels v5.19+, we use bpf_ringbuf_reserve_dynptr. This allows reserving a variable amount of space determined at runtime (e.g., the actual length of the string).
3. **Fallback:** For older kernels, we use a "Max Size" strategy. We reserve a chunk equal to MAX_PATH (or a reasonable limit like 512 bytes). We write the string into this space. If the string is shorter, we submit the full chunk, and userspace relies on the filename_len in the header to read only the valid bytes. While this wastes some ring buffer space (internal fragmentation), it avoids the double-copy overhead of bpf_ringbuf_output (which requires copying data to the stack first).[6]

3.4 Userspace Architecture: C++ RAII

The userspace component handles the lifecycle of the eBPF programs using C++ RAII patterns. This ensures that BPF resources are cleaned up deterministically, even in the event of exceptions.

The BpfSkeleton Wrapper:

C++

```
class BpfSkeleton {
public:
    // Constructor handles loading and attachment
    BpfSkeleton(const std::string& filename) {
        skel = aegis_bpf__open();
        if (!skel) throw std::runtime_error("Failed to open BPF skeleton");

        // Configure global variables before load if necessary
        skel->rodata->config_pid = getpid();

        if (aegis_bpf__load(skel)) {
            aegis_bpf__destroy(skel);
            throw std::runtime_error("Failed to load BPF skeleton");
        }

        if (aegis_bpf__attach(skel)) {
            aegis_bpf__destroy(skel);
            throw std::runtime_error("Failed to attach BPF programs");
```

```
        }

        // Initialize Ring Buffer Manager
        rb = ring_buffer__new(bpf_map__fd(skel->maps.events), handle_event, nullptr, nullptr);
    }

    // Destructor ensures cleanup
    ~BpfSkeleton() {
        ring_buffer__free(rb);
        aegis_bpf__destroy(skel);
    }

    void poll() {
        // Efficient polling using epoll under the hood
        ring_buffer__poll(rb, 100 /* timeout ms */);
    }

private:
    struct aegis_bpf* skel;
    struct ring_buffer* rb;

    static int handle_event(void *ctx, void *data, size_t data_sz) {
        // Event processing logic
        return 0;
    }
};
```

This wrapper abstracts the complexity of libbpf's lifecycle functions (_open, _load, _attach, _destroy) into a safe C++ object. When the agent shuts down, the destructor automatically detaches probes and unloads the program.[20]

3.5 Core Hook Selection

AegisBPF targets a minimal set of high-value hooks to balance visibility with performance:

Hook Point	Tracepoint/Probe	Purpose
Execution	tracepoint/syscalls/sys_enter_execve	Capture binary execution arguments (argv).
Execution	tracepoint/syscalls/sys_exit	Capture return values (did

	_execve	the execution succeed?).
Fork/Clone	tracepoint/sched/sched_process_fork	Track parent-child relationships and update process_tree map.
Exit	tracepoint/sched/sched_process_exit	Clean up process_tree map entries to prevent leaks.
File Access	tracepoint/syscalls/sys_enter_openat	Monitor file opening. Used with Bloom filter for enforcement.
Networking	tracepoint/syscalls/sys_enter_connect	Monitor outbound network connections.
Helper	kprobe/fd_install	Map a File Descriptor (returned by open) to a Filename. This is crucial because subsequent write calls only use the FD, obscuring the target file.[13]

4. Implementation Challenges and Solutions

4.1 The "Write" Blindness Problem

A common pitfall in eBPF security is monitoring write syscalls. A write(fd, buffer) call only provides a file descriptor integer. It does not tell you which file is being written to.
Solution: AegisBPF hooks fd_install (or sys_enter_openat return). When a file is opened, we map the returned fd + pid to the filename in a fd_cache map. When a write occurs, we look up the fd in this map to retrieve the filename.

4.2 Kernel Version Compatibility

AegisBPF utilizes CO-RE. However, older kernels (pre-5.8) lack the Ring Buffer.
Solution: The userspace agent checks kernel features at startup. If BPF_MAP_TYPE_RINGBUF is unavailable, it falls back to BPF_MAP_TYPE_PERF_EVENT_ARRAY. The BPF C code uses macro definitions to conditionally compile the correct map type, or utilizes libbpf's map-in-map capabilities to abstract this.4

5. Conclusion

The transition from Falco's userspace evaluation to Tetragon's kernel-native enforcement

marks a maturation of the Linux security ecosystem. While Falco remains excellent for broad, compatibility-focused auditing, the future lies in synchronous, inline prevention.

AegisBPF synthesizes the best practices of this generation:

1. **Tracee's** rigorous handling of PID recycling via start-time tracking.
2. **Tetragon's** inline enforcement model using signals.
3. **Modern C++** engineering to provide a safe, high-performance userspace runtime without the overhead of garbage collection.

By leveraging advanced structures like Bloom filters for enforcement and Ring Buffers for transport, AegisBPF represents a theoretical design that addresses the specific pain points—overhead, latency, and identity reuse—that plague current implementations. As eBPF capabilities like the "BPF Arena" and signed policies mature, this architecture is positioned to evolve into a secure root-of-trust for the entire operating system.

Tables and Data Representation

Table 1: Kernel Hook Comparison across Tools

Tool	Primary Hook Type	Key Hooks Utilized	Enforcement Mechanism
Falco	Tracepoints (Raw)	sys_enter, sys_exit	None (Userspace sidekick)
Tracee	Tracepoints + Kprobes	vfs_write, security_file_open	None (Forensic focus)
Tetragon	Kprobes	fd_install, do_execve, tcp_connect	bpf_send_signal, bpf_override_return
AegisBPF	Tracepoints + Kprobes	sched_process_fork, sys_enter_execve	bpf_send_signal (via Bloom Filter)

Table 2: Overhead Comparison (Estimated)

Metric	Userspace Analysis (Falco)	In-Kernel Filtering (Tetragon/AegisBPF)

CPU Overhead	5-15% (High syscall volume)	< 1-2%
Latency	Milliseconds (Async)	Microseconds (Sync)
Event Volume	High (All events sent to user)	Low (Filtered in kernel)
Prevention	Impossible (TOCTOU)	Possible (Inline)

Works cited

1. What is eBPF? An Introduction and Deep Dive into the eBPF Technology, accessed on January 11, 2026, https://ebpf.io/what-is-ebpf/
2. eBPF - Introduction, Tutorials & Community Resources, accessed on January 11, 2026, https://ebpf.io/
3. Hook points | Tetragon - eBPF-based Security Observability and Runtime Enforcement, accessed on January 11, 2026, https://tetragon.io/docs/concepts/tracing-policy/hooks/
4. Kernel Events | Falco, accessed on January 11, 2026, https://falco.org/docs/concepts/event-sources/kernel/
5. Kernel Events Architecture | Falco, accessed on January 11, 2026, https://falco.org/docs/concepts/event-sources/kernel/architecture/
6. Map Type 'BPF_MAP_TYPE_RINGBUF' - eBPF Docs, accessed on January 11, 2026, https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_RINGBUF/
7. Falco vs. Tetragon: A Runtime Security Showdown for Kubernetes | by Asim Mirza | Medium, accessed on January 11, 2026, https://medium.com/@mughal.asim/falco-vs-tetragon-a-runtime-security-showdown-for-kubernetes-a0e9fb9f30a0
8. Container Runtime Security Tooling Comparison - AccuKnox, accessed on January 11, 2026, https://accuknox.com/wp-content/uploads/Container_Runtime_Security_Tooling.pdf
9. Architecture - Tracee - Aqua Security, accessed on January 11, 2026, https://aquasecurity.github.io/tracee/v0.8.3/architecture/
10. An In-Depth Analysis of eBPF-Based System Security Tools in Cloud-Native Environments - IEEE Xplore, accessed on January 11, 2026, https://ieeexplore.ieee.org/iel8/6287639/10820123/11146725.pdf
11. Ordering Events - Tracee - Aqua Security, accessed on January 11, 2026, https://aquasecurity.github.io/tracee/v0.11/docs/deep-dive/ordering-events/
12. Trace Options - Tracee - Aqua Security, accessed on January 11, 2026, https://aquasecurity.github.io/tracee/v0.5.1/tracee-ebpf/trace-options/
13. Securing the Modern Process with Tetragon: Runtime Security for the

Cloud-Native Kernel, accessed on January 11, 2026, https://cilium.io/blog/2025/11/4/process-tetragon/
14. Tetragon - eBPF-based Security Observability and Runtime Enforcement, accessed on January 11, 2026, https://tetragon.io/
15. Pre-exploit mitigation · Issue #4399 · cilium/tetragon - GitHub, accessed on January 11, 2026, https://github.com/cilium/tetragon/issues/4399
16. bpf_send_signal - Engineering Everything with eBPF, accessed on January 11, 2026, https://ebpf.hamza-megahed.com/docs/chapter5/4-bpf_send_signal/
17. Events | Tetragon - eBPF-based Security Observability and Runtime Enforcement, accessed on January 11, 2026, https://tetragon.io/docs/concepts/events/
18. Security superpowers with eBPF and Tetragon - Cisco Live, accessed on January 11, 2026, https://www.ciscolive.com/c/dam/r/ciscolive/global-event/docs/2025/pdf/BRKSEC-2167.pdf
19. eBPF Explained: Use Cases, Concepts, and Architecture | Tigera - Creator of Calico, accessed on January 11, 2026, https://www.tigera.io/learn/guides/ebpf/
20. How to Build Portable eBPF Programs with CO-RE, accessed on January 11, 2026, https://oneuptime.com/blog/post/2026-01-07-ebpf-core-portable-programs/view
21. C++ Beyond the Syllabus #4: RAII & Smart Pointers | by Jared Miller - Medium, accessed on January 11, 2026, https://jaredmil.medium.com/c-beyond-the-syllabus-4-raii-smart-pointers-7c784134ace5
22. Hello eBPF: Ring buffers in libbpf (6) - foojay, accessed on January 11, 2026, https://foojay.io/today/hello-ebpf-ring-buffers-in-libbpf-6/
23. libbpf Overview — The Linux Kernel documentation, accessed on January 11, 2026, https://www.kernel.org/doc/html/v6.6/bpf/libbpf/libbpf_overview.html
24. Getting to Know TGID and PID in eBPF: Essential for Observability - Yuki Nakamura's Blog, accessed on January 11, 2026, https://yuki-nakamura.com/2025/02/15/getting-to-know-tgid-and-pid-in-ebpf-essential-for-observability/
25. Map Type 'BPF_MAP_TYPE_BLOOM_FILTER' - eBPF Docs, accessed on January 11, 2026, https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_BLOOM_FILTER/
26. BPF_MAP_TYPE_BLOOM_FILT... - The Linux Kernel documentation, accessed on January 11, 2026, https://docs.kernel.org/bpf/map_bloom_filter.html
27. map_bloom_filter.rst, accessed on January 11, 2026, https://www.kernel.org/doc/Documentation/bpf/map_bloom_filter.rst
28. BPF ring buffer - The Linux Kernel documentation, accessed on January 11, 2026, https://docs.kernel.org/bpf/ringbuf.html
29. libbpf documentation, accessed on January 11, 2026, https://libbpf.readthedocs.io/_/downloads/en/v1.4.3/pdf/