Ironclad Kernel

The Next-Generation eBPF Security Platform

Technical Whitepaper & Architecture Guide

Architecture: Split-Plane (eBPF Data Plane + Go Control Plane)

November 26, 2025

Abstract

Executive Summary: Ironclad Kernel addresses the limitations of traditional perimeter firewalls by moving security enforcement directly into the Linux Kernel using **eBPF** (Extended Berkeley Packet Filter). This approach enables identity-aware micro-segmentation, zero-trust enforcement, and deep observability of encrypted traffic (TLS) without the performance penalties of userspace proxies.

Contents

1 Introduction

In the modern threat landscape, adversaries bypass Layer 3/4 controls using legitimate ports (80/443). Once inside, they utilize standard binaries for lateral movement and C2 channels. Ironclad Kernel mitigates this by enforcing security policies based on **Process Identity** (comm, PID) rather than just IP addresses.

2 Core Architecture

Ironclad operates on a high-performance **Split-Plane Architecture**.

2.1 The Data Plane (Kernel Space)

The enforcement engine runs entirely within the Linux Kernel as JIT-compiled eBPF bytecode. This ensures tamper resistance and minimal latency.

2.1.1 Key Kernel Hooks

Hook Point	Functionality
`cgroup/connect4`	**Primary Enforcement.** Triggers on every TCP/UDP connection attempt (`connect()` syscall). Blocks malicious traffic before a SYN packet is generated.
`uprobe/SSL_write`	**TLS Visibility.** Hooks into OpenSSL libraries to capture plaintext data before encryption, enabling inspection without MitM certificates.
`kprobe/tcp_sendmsg`	**Telemetry.** Captures traffic volume for bandwidth accounting and DNS query inspection at the socket layer.

Table 1: eBPF Instrumentation Points

2.2 The Control Plane (User Space)

A high-concurrency **Go daemon** manages the lifecycle of eBPF programs.

- **Map Management:** Dynamically updates Kernel Maps (Hash Maps, LPM Tries) with rules and threat feeds.
- **Event Stream:** Consumes the **Ring Buffer** shared memory structure for high-performance logging without `perf_event` overhead.
- **Threat Intelligence:** Asynchronously fetches feeds (e.g., Abuse.ch) and pushes compiled binary keys to the kernel.

3 Threat Mitigation Capabilities

3.1 Identity-Based Zero Trust

Unlike traditional firewalls that see packets, Ironclad sees processes.

- **Context:** ”Process `/usr/bin/curl` (PID 1234) attempting to reach 8.8.8.8”.

- **Policy:** Allows `apt-get` to update while blocking malicious scripts attempting the same connection.

3.2 Geo-Fencing with LPM Tries

Blocking entire countries requires checking IPs against thousands of CIDR ranges.

- **Challenge:** Linear scanning is $O(N)$ and too slow for the kernel.
- **Solution:** Ironclad uses **Longest Prefix Match (LPM) Trie** maps.
- **Performance:** Lookup complexity is $O(1)$ relative to rule count (bound by 32-bit key length), enabling 50,000+ subnet blocks with nanosecond latency.

3.3 TLS Inspection ("God Mode")

Ironclad solves the "Dark Data" problem of encrypted C2 channels.

- **Technique:** User Probes (`uprobes`) attached to `SSL_write` in `libssl.so`.
- **Result:** Captures full HTTP requests (Headers, URL, Body) in plaintext.
- **Privacy:** Data is streamed only to the local Ring Buffer for analysis; it is never stored externally.

3.4 Volumetric DDoS Protection

- **Mechanism:** Token Bucket rate limiter in eBPF maps.
- **Enforcement:** Drops `connect()` syscalls immediately if the rate exceeds thresholds (e.g., 100 conn/s), preventing local resource exhaustion.

4 Performance & Specifications

Designed for high-density Kubernetes nodes and Edge devices.

4.1 Benchmarks (Estimated)

- **CPU Overhead:** $< 1\%$ increase at 10Gbps throughput.
- **Latency:** < 500 nanoseconds added per connection attempt.
- **Memory Footprint:** $\sim$20MB (Daemon) + $\sim$50MB (Kernel Maps).

4.2 Operational Excellence

- **Boot Race Protection:** Uses `After=network-online.target` to prevent startup failures before threat feeds download.
- **Fail-Safe Mode:** If the Userspace Daemon crashes, eBPF programs remain active in the kernel, persisting protection.
- **Single Binary:** Statically linked (Daemon + CLI + Web UI + Bytecode). No external dependencies like Python or Libpcap.
- **Unified Config:** Managed via a single YAML file: `/etc/go-ebpf-firewall/rules.yaml`.