
VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

The research of high-level tracing for Linux extended Berkeley Packet Filter

# In-Kernel Packet Sanitization and Normalization Using eBPF

Done by:
Mantas Jonaitis signature

Supervisor:
dr. Linas Bukauskas

Vilnius
2026

# Contents

# Anotacija

**Tinklo paketų valymas ir normalizavimas Linux branduolyje naudojant eBPF ir XDP technologijas**

Tradiciniai paketų apdorojimo sprendimai susiduria su našumo ir lankstumo kompromisais, o egzistuojančios tinklo paketų filtravimo sistemos daugiausia yra orientuotos į paketų praleidimą arba atmetimą, o ne filtravimą ar sanitizavimą. Šiandieniniai naujausi pasiekimai Linux branduolio eBPF posistemejė leidžia kurti programas, kurios gali efektyviai apdoroti didelius kiekius tinklo paketų pačiame Linux branduolyje, nekuriant sudėtingų Linux branduolio modulių ir nereikalaujant branduolio perkrovimo. Šis baigiamasis darbas pristato moderniomis Linux branduolio technologijomis eBPF ir XDP pagrįstos aukšto našumo tinklo paketų apdorojimo, filtravimo ir valymo sistemos architektūrą ir implementacijos apžvalgą. Sukurta paketų valymo sistema susideda iš kelių pagrindinių komponentų. Go programavimo kalba parašytos vartotojo erdvės programos, kuri veikia, kaip valdymo ir konfigūracijos sąsaja, užkrauna eBPF programas į branduolį, apdoroja vartotojo apibrėžtas tinklo paketų valymo taisykles, atlieka įvairias stebėjimo funkcijas ir pateikia statistikas apie apdorotus, išvalytus ar atmestus tinklo paketus. Branduolio erdvėje veikianti eBPF programa yra užkraunama XDP taške, kuris leidžia apdoroti tinklo paketus iš karto po arba net pačioje tinklo plokštėje - dar prieš paketams pasiekiant tradicinius Linux branduolio tinklo apdorojimo sluoksnius. Tai leidžia labai efektyviai ir greitai atlikti tinklo paketų filtravimą ir valymą, išvengiant brangių kontekstų perjungimų. Sukurtas sprendimas buvo eksperimentaliai įvertintas su skirtingais tinklo paketų srauto kiekiais nuo 1 iki 500 tūkstančių paketų per sekundę, įvertinant visas implementuotas paketų valymo funkcijas. Testavimo aplinkoje generuoti dirbtini tinklo paketai buvo apdorojami naudojant sukurtą programą ir įvairius kiekius valymo taisyklių. Eksperimentinio tyrimo rezultatai parodė, kad XDP/eBPF tinklo paketų valymo programos implementacija yra pajėgi apdoroti paketus maksimaliu tinklo srauto greičiu. Įvairių paketų valymo taisyklių apdorojimo laikas išliko beveik pastovus ir kito nežymiai skirtinguose tinklo srauto kiekiuose. Paketų apdorojimo greitis priklauso nuo naudojamų taisyklių kiekio ir jų sudėtingumo. Lyginant su labiau tradiciniu Linux branduolio posistemės Netfilter NFQUEUE sprendimu, XDP/eBPF implementacijos pajėgumas buvo beveik penkis kartus didesnis, bei atlaikė visą generuotų paketų srautą. Sukurta sistema palaiko TCP ir UDP tinklo protokolų paketus, tačiau yra lengvai praplečiama norint palaikyti kitus protokolus. Protokolų apdorojimo funkcijos yra modulinės ir lengvai išplečiamos. Dabartinė implemntacija veikia pagrinde su paketų antraštėmis - norint atlikti sudėtingėsnius paketų analizės ir apdorojimo mechanzimus reikėtų išplėsti efektyvaus paketų perkėlimo iš branduolio į vartotojo erdvę mechanizmus naudojantis `AF_XDP` lizdų technologija. Šio darbo rezultatai rodo, jog siūlomas Linux branduolio eBPF ir XDP technologijomis grįstas sprendimas suteikia efektyvų kompromisą tarp našumo ir lankstumo tinklo paketų filtravimo ir valymo uždaviniams spręsti ir galėtų būti sėkmingai pritaikytas realiose tinklo saugumo sistemose.

Raktiniai žodžiai: eBPF, XDP, tinklo paketų valymas, tinklo paketų filtravimas, Linux tinklo posistemė.

# Summary

Traditional packet-processing solutions face trade-offs between performance and flexibility, while existing network packet filtering systems are largely oriented toward allowing packets through or dropping them rather than filtering or sanitizing them. Recent advances in the Linux kernel's eBPF subsystem enable the development of programs that can efficiently process large volumes of network traffic directly within the Linux kernel, without building complex kernel modules and without requiring a kernel reboot. This thesis presents an overview of the architecture and implementation of a high-performance network packet processing, filtering, and sanitization system based on modern Linux kernel technologies-eBPF and XDP.

The developed packet sanitization system consists of several core components. A user-space application written in Go acts as the management and configuration interface: it loads eBPF programs into the kernel, processes user-defined packet sanitization rules, performs various monitoring functions, and provides statistics on processed, sanitized, or dropped network packets. The kernel-space eBPF program is attached at the XDP hook, which enables packet processing immediately after reception-or even on the network interface card-before packets reach the traditional Linux networking stack. This allows network packet filtering and sanitization to be performed very efficiently and quickly, avoiding expensive context switches.

The solution was experimentally evaluated under a range of traffic rates from 1 to 500 thousand packets per second, covering all implemented packet sanitization functions. In the test environment, synthetic network packets were processed using the developed program with varying numbers of sanitization rules. The experimental results showed that the XDP/eBPF packet sanitization implementation is capable of processing packets at the maximum traffic rate. The processing time of different sanitization rules remained nearly constant and varied only slightly across different traffic loads. Packet processing throughput depends on the number of rules used and their complexity. Compared with the more traditional Linux subsystem solution based on Netfilter NFQUEUE, the XDP/eBPF implementation achieved nearly five times higher throughput and sustained the full generated packet stream.

The developed system supports TCP and UDP traffic, but can be easily extended to support additional protocols. The protocol-processing functions are modular and straightforward to expand. The current implementation operates primarily on packet headers; to perform more advanced packet analysis and processing mechanisms, it would be necessary to extend efficient mechanisms for transferring packets from kernel space to user space, for example by using `AF_XDP` sockets. Overall, the results indicate that the proposed solution based on Linux kernel eBPF and XDP technologies provides an effective compromise between performance and flexibility for network packet filtering and sanitization tasks and could be successfully applied in real-world network security systems.

Keywords: eBPF, XDP, network packet sanitization, network packet filtering, Linux networking stack.

# Glossary

**AF_XDP** Address Family eXpress Data Path - A high-performance socket interface providing zero-copy packet I/O between user-space and XDP for fast packet processing.

**BPF Maps** Kernel data structures enabling communication between eBPF programs and user-space, supporting hash tables, arrays, ring buffers, and per-CPU variants.

**BPF Verifier** Static analysis component ensuring eBPF program safety by checking memory access patterns, loop bounds, and instruction validity before loading.

**bpf2go** Code generation tool from Cilium eBPF library that compiles BPF C code and generates Go bindings for integration.

**eBPF** extended Berkeley Packet Filter - In-kernel virtual machine allowing safe execution of user-defined programs for networking, tracing, and security without kernel modifications.

**eBPF Helper Functions** Kernel-provided functions that eBPF programs can safely call for operations like memory access, packet manipulation, or map updates.

**First-Match Semantics** Rule evaluation strategy where packet processing stops after the first matching rule is applied, used in this thesis's rule engine.

**JIT Compilation** Just-In-Time compilation of eBPF bytecode to native machine code for improved performance.

**NFQUEUE** Netfilter Queue - Linux kernel facility allowing packets to be passed to user-space applications for processing and verdict determination.

**Packet Sanitization** Process of modifying malicious or malformed packets to prevent security vulnerabilities while allowing traffic to continue (distinct from simple drop/pass).

**Per-CPU Map** BPF map type where each CPU core maintains its own data copy, avoiding synchronization overhead and lock contention.

**sk_buff** Linux kernel data structure representing a network packet with metadata and pointers; allocated later in the stack than XDP processing.

**Tail Call** BPF mechanism allowing one BPF program to call another without increasing stack depth, enabling modular program design with protocol-specific handlers.

**XDP** eXpress Data Path - High-performance programmable packet processing framework operating at the earliest point in the Linux network stack, before sk_buff allocation.

**XDP_DROP / XDP_PASS / XDP_REDIRECT** XDP verdict return codes: DROP discards packets immediately, PASS continues to kernel stack, REDIRECT forwards to AF_XDP or another interface.

**IDS** Intrusion detection system.

**IPS** Intrusion prevention system.

**NIC** Network interface card.

# Introduction

Modern high-speed networked systems demand packet processing mechanisms that operate at network line rate without compromising flexibility, speed, or security. Existing approaches, such as kernel-bypass or userspace offloading frameworks, face inherent trade-offs: they either sacrifice security or flexibility for performance, introduce complexity through external dependencies, or lack security or functional features that are provided by the kernel and its networking stack. Existing traditional kernel-space solutions are complex and generalized, buried within layers and layers of abstraction, performing critical packet filtering decisions too late in the packet data path and making them too slow for particular packet processing flows. Traditional IDS/IPS systems or firewalls focus mostly on dropping or passing packets to the next layer, and packet normalization with sanitization is not a widely explored topic with only a few existing solutions that are either not functional enough or are outdated. In-kernel packet sanitization and normalization offers a compelling addition to the binary drop/pass mechanisms prevalent in existing firewalls and intrusion detection systems.

Recent developments and advancements in Linux kernel technologies, especially eBPF (extended Berkeley Packet Filter) and XDP (eXpress Data Path), have opened new avenues for high-performance packet processing and modification directly within the kernel boundary. eBPF allows for safe, efficient, and flexible execution of custom code in the kernel space, while XDP framework enables to process packets within the safe context of eBPF program at the earliest point in the kernel networking stack without passing packets through deeper layers of networking stack. These technologies provide an opportunity to build packet processing systems that can achieve line-rate performance while maintaining the flexibility and security features of kernel-space.

This master thesis aims to develop a high-performance, highly configurable in-kernel packet sanitization engine using eBPF and XDP technologies. This aim leads to the following main tasks of this work:

- Analyze existing literature on packet processing and filtering, find similar projects and overview currently prevalent solutions and techniques available for packet sanitization in other systems.
- Explore the obscure and not widely researched topic of packet sanitization.
- Design and implement a performant and highly configurable in-kernel packet sanitization system using eBPF and XDP frameworks.
- Build a lean testing framework for generating synthethic network packet traffic and excercising that traffic against the developed packet sanitizer system.
- Evaluate the performance and effectiveness of the developed system.
- Using Netfilter NFQUEUE subsystem, develop a small functionally representative piece of the developed eBPF/XDP packet sanitizer for quantitative performance comparison of in-kernel and user-space programs in packet sanitization.

The paper is structured as follows. Firstly in Section 1, we present a literature overview of existing technologies and research related to packet processing, filtering and sanitization/normalization techniques. In Section 2 we introduce a detailed architectural overview of the full in-kernel

packet sanitization system, its management plane, rule system and packet processing and sanitization logic. We detail the implementation features and available sanitization rules. Finally in Section 3 we present the numerical evaluation of the implemented eBPF packet sanitizer performance, compare it with Netfilter NFQUEUE based approach and discuss the results.

# 1 Literature review

## 1.1 Traditional packet processing in the Linux kernel

Traditional packet filtering in Linux relies primarily on the Netfilter [8] subsystem, which provides a set of hooks within the kernel networking stack for intercepting and processing packets. The Netfilter architecture defines five main hook points (`NF_IP_PRE_ROUTING`, `NF_IP_LOCAL_IN`, `NF_IP_FORWARD`, `NF_IP_LOCAL_OUT`, and `NF_IP_POST_ROUTING`) that allow rule-based processing at different stages of packet traversal through the network stack. This hook-based model enables both stateless filtering based on packet headers and stateful connection tracking that maintains session information for connection-oriented protocols such as TCP. The connection tracking (conntrack) subsystem of Netfilter maintains state information for active network connections, enabling stateful packet filtering and Network Address Translation (NAT). However, this stateful processing comes with substantial memory overhead [7], as the kernel must store connection state for each active connection. In high-throughput environments, the conntrack table can become a memory and processing bottleneck, particularly when handling large numbers of concurrent connections or connection-heavy protocols. The iptables and its successor nftables serve as user-space interfaces to the Netfilter subsystem, allowing administrators to define filtering rules based on IP addresses, ports, protocols, and connection states. These rules are organized into chains and are evaluated sequentially per packet. This linear processing of rules can lead to linear performance degradation as the number of rules increases [23].

The limitations of traditional Netfilter-based filtering become apparent in scenarios requiring complex packet analysis or modification, or when the number of rules increases drastically. While Netfilter excels at header-based filtering decisions, by default, it provides limited programmability for custom packet processing logic. The rule syntax, while flexible for common use cases, cannot easily express complex sanitization policies or protocol-specific validation or packet contents modification logic.

To address some of these limitations, mechanisms like libnetfilter_queue `NFQUEUE` allow packets to be diverted to user-space applications for more sophisticated processing. While NFQUEUE enables complex packet inspection and modification capabilities beyond Netfilter's native functionality, this approach introduces performance overhead due to kernel-to-userspace context switching and packet copying [21][6]. The additional latency and reduced throughput make NFQUEUE unsuitable for high-performance packet sanitization scenarios where maintaining line-rate processing is critical.

These characteristics suggest that while traditional Netfilter-based solutions are effective for many use cases, complex packet sanitization scenarios requiring both high performance and flexible processing logic may benefit from a more specialized solution that can provide kernel-level performance while maintaining the flexibility required for advanced sanitization policies.

## 1.2 OpenBSD PF firewall

OpenBSD's Packet Filter (PF) emerged in 2001 as the successor to IPFilter, providing the project with a firewall that matched its secure-by-default philosophy and auditable code base [24]. PF adopts a policy-driven rule language that separates filtering, translation, and queuing, enabling administrators to express complex behaviours succinctly while keeping the kernel implementation compact. Early design decisions such as anchored rule sets, first-match semantics, and pervasive

state tracking were intended to mitigate common firewall misconfigurations and to reduce the risk of ambiguous packet handling paths [19]. These characteristics made PF attractive not only on OpenBSD but also in derivative operating system distribution projects such as FreeBSD, NetBSD, and macOS, where PF continues to ship as the default packet filter for many deployments.

PF's stateful inspection engine maintains per-flow metadata that records connection direction, TCP sequence windows, and policy decisions. By default the firewall synthesises state entries for both TCP and UDP sessions, allowing return traffic to traverse without re-evaluating the rule set while enabling robust tracking primitives such as `sloppy state` for asymmetric routing scenarios [25]. The same state table acts as a shared information bus for ancillary features, including traffic shaping via ALTQ or the packet prioritisation framework, thereby tying enforcement and resource allocation to consistent packet classifications [19]. This tight integration between policy and state is a defining difference from iptables/nftables, where connection tracking is an optional module whose metadata must be queried explicitly.

The normalisation subsystem invoked through `scrub` rules or via implicit defaults is PF's answer to traffic ambiguity and protocol stacks that intentionally violate RFC specifications. When a packet matches a normalisation rule, PF rewrites header fields to conservative values, strips dangerous options, and attempts reassembly so that downstream policy decisions operate on canonical packet representations [27]. This behaviour addresses well-known intrusion detection evasion techniques that rely on inconsistent fragmentation handling or TCP flag manipulation. PF can drop or rewrite overlapping IP fragments, randomise IPv4 identifiers to prevent traffic correlation, and enforce minimum IPv4 Time-To-Live (TTL) values to defeat hop-count-based covert channels [26]. The scrubber also reassembles IPv4 and IPv6 fragments when requested, enabling state tracking and deep inspection engines both inside and outside the kernel to analyse complete payloads rather than ambiguous fragments.

PF exposes normalisation controls through a concise syntax. A single `scrub` rule can apply `no-df` to clear the Do Not Fragment bit, enforce `max-mss` to rewrite TCP Maximum Segment Size values that exceed path MTU assumptions, or set `min-ttl` to drop packets that arrive with suspiciously low hop counts [27]. Administrators can combine these directives with stateful options such as `max-mss 1440 fragment reassemble` to ensure dedicated flows remain standards-compliant even when remote peers exhibit non-conformant behaviour. PF further extends normalisation into the policy layer: `match` rules allow administrators to perform conditional rewrites before filtering decisions occur, while `set state-policy` controls dictate whether the firewall `modulate state` by randomising TCP sequence numbers or `synproxy state` to complete TCP handshakes on behalf of internal services [19].

PF's design treats normalisation as an integral part of the security posture rather than an optional bolt-on. Because scrub decisions are executed in the same kernel path as filtering, administrators avoid the performance penalties arising from diverting packets to user space for sanitisation, a common requirement when replicating similar policies with Netfilter and NFQUEUE [21]. Tight coupling also means normalised packets are recorded in the state table after their transformation, maintaining consistency between what the firewall observes and what back-end services receive. This alignment is particularly valuable for network intrusion detection systems such as Snort or Suricata that monitor mirrored traffic: the firewall eliminates ambiguities before packets reach downstream sensors, reducing the burden on those tools to implement their own reassembly logic.

Although PF operates primarily on OpenBSD, its normalisation features have influenced the design of modern packet-processing frameworks. FreeBSD's port of `pf` retains `scrub` semantics

to protect edge services, and vendors of firewall appliances continue to rely on PF's normalisation primitives when marketing "protocol sanitisation" features. In the context of this thesis, PF serves as an established benchmark for kernel-resident sanitisation: it demonstrates how carefully curated modifications such as TTL clamping or MSS rewriting can be executed at line rate while maintaining a maintainable policy language. Comparing PF's scrub capabilities with Linux-based alternatives helps identify functional gaps that programmable data paths such as eBPF/XDP must bridge to offer comparable resilience against evasive or malformed traffic.

## 1.3 User space networking frameworks

To overcome the performance limitations of kernel-based packet processing, several user-space frameworks have emerged that bypass the traditional kernel networking stack entirely. Initially developed by Intel, the Data Plane Development Kit or DPDK [9] represents the most prominent approach in this category, providing a set of libraries and drivers for fast packet processing. DPDK achieves high performance through several key architectural decisions: poll-mode drivers (PMDs) that eliminate interrupt-driven processing, user-space drivers that bypass kernel involvement, and direct memory access to network interface card (NIC) buffers through technologies like UIO (Userspace I/O) and VFIO (Virtual Function I/O).

The performance benefits of DPDK are substantial. By eliminating kernel-to-userspace context switches and bypassing the kernel network stack and implementing zero-copy packet handling, DPDK applications can achieve line-rate processing even over 100 milion packets per second and beyond on modern commodity hardware [20] [17]. The poll-mode approach, while CPU-intensive, eliminates the latency variability associated with interrupt processing present in traditional kernel networking stack and provides predictable performance characteristics essential for real-time packet processing applications.

Alternative user-space frameworks provide similar capabilities with different design trade-offs. The netmap framework [28] offers a more lightweight approach by providing efficient packet I/O primitives while maintaining some integration with the kernel networking stack. PF_RING [13] focuses on high-speed packet capture and filtering, particularly suited for network monitoring applications. Intel's specialized networking libraries complement DPDK by providing optimized implementations of common networking functions and protocol stacks.

However, while providing exceptional processing speeds, these user-space solutions introduce significant integration complexity and operational trade-offs. Applications utilizing kernel-bypass frameworks lose access to standard kernel networking features and integration with OS networking tools. This isolation requires applications to reimplement substantial portions of networking functionality, increasing development complexity and maintenance burden. Additionally, DPDK applications typically require dedicated CPU cores and memory allocation, creating resource isolation requirements that can complicate deployments in multi-tenant or containerized environments.

The deployment challenges extend beyond technical complexity to operational considerations. DPDK applications require specialized configuration for hugepage memory allocation, CPU core assignment, and NIC binding to user-space drivers. These requirements make it difficult to integrate high-performance packet processing into existing network infrastructure without substantial architectural changes. Furthermore, the all-or-nothing nature of kernel bypass means that applications cannot selectively apply high-performance processing to specific traffic patterns while maintaining standard kernel handling for other traffic.

These characteristics position user-space high-performance frameworks as powerful but spe-

cialized solutions that excel in dedicated packet processing applications but struggle with integration into general-purpose networking environments where flexible, selective packet processing is required.

## 1.4 Packet Processing in Network Security Applications

Network security applications employ diverse packet processing architectures with varying degrees of modification capabilities, yet most operate with fundamental limitations in selective packet sanitization.

Snort's [10] packet processing occurs in user-space through libpcap or AF_PACKET interfaces by default, requiring kernel-to-userspace packet copying for each inspection operation. While Snort can perform basic packet normalization through preprocessors, its modification capabilities are limited to simple header rewriting and cannot perform complex sanitization operations [29]. Additionally, Snort operates in single-threaded mode, which limits its performance and ability to handle high loads during an increased volume of malicious traffic [30].

Suricata extends traditional IDS functionality with inline processing mode to process packets in the direct packet hot-path, making it a full-fledged intrusion prevention system, or IPS for short. It supports multiple packet acquisition methods including NFQUEUE, AF_PACKET, and AF_XDP for varying performance profiles [5]. While AF_XDP integration reduces some kernel-userspace overhead, traditional NFQUEUE deployments still suffer from context switching and packet copying penalties. Suricata's multi-threaded architecture and eBPF support for traffic filtering provide performance advantages over single-threaded solutions [12]. However, packet modification capabilities remain limited to basic header rewriting, payload replacement, and rule-based actions, with complex sanitization operations requiring custom Lua scripting that introduces significant performance overhead [1].

The fundamental limitation across existing approaches is the trade-off between processing flexibility and performance. User-space solutions like Snort and Suricata provide programmable packet modification capabilities but suffer from kernel-userspace packet copying overhead and context switching latency. Furthermore, current packet modification approaches exhibit redundant processing overhead. Packets often traverse multiple inspection stages: first through kernel filtering, usually via the Netfilter subsystem, then via user-space IDS/IPS programs, with each stage requiring separate packet parsing and state maintenance. This architectural redundancy compounds latency and reduces overall system throughput. In-kernel packet sanitization using eBPF could address these limitations by combining the performance benefits of kernel-space processing with the programmability required for complex sanitization operations, while eliminating redundant packet traversal through multiple processing layers.

## 1.5 eBPF - the extended Berkeley Packet Filter

The extended Berkeley Packet Filter (eBPF) is a recent successor to the initially novel idea of BPF [22]. First introduced by Alexei Starovoitov, eBPF has evolved into a comprehensive in-kernel virtual machine that enables safe execution of user-defined programs without kernel modifications [15]. The eBPF verifier ensures eBPF programs are safe before executing them. It checks memory access, prevents loading programs with infinite loops, and verifies all code paths terminate properly. Loaded eBPF program bytecode is executed within an in-kernel eBPF virtual machine. The eBPF virtual machine is an eleven-register stack-based architecture with an instruction set that is designed

to be simple and efficient [15]. The programs have bounded memory access and can only call approved helper functions; therefore, there is no C standard library available when writing C eBPF programs. The eBPF VM can perform JIT compilation to convert the eBPF VM bytecode to native machine code for even better performance. Communication between user-space and kernel-space in eBPF is performed via a storage facility called a BPF map [2]. BPF maps support different data structures such as hash tables, arrays, ring buffers, and others [2]. Maps can be polled from user-space by conventional polling mechanisms calling the `bpf` system call or via efficient notification facilities such as `epoll` for supported map types such as `BPF_MAP_TYPE_RINGBUF` [3].

There are multiple tracepoints and hooks, also referred to as "designated code paths" [15], where eBPF programs can be attached for not only packet processing but also syscall and other functionality. Depending on when we want to process a packet in the packet data path, an eBPF program can be attached at: XDP (eXpress Data Path) hook in the NIC or just after NIC receive queue packet processing; TC (Traffic Control) hook for processing packets in the queueing discipline when `sk_buff` is allocated; socket level and other hooks. XDP operates at the earliest possible point in the packet receive path, directly accessing the driver's receive ring buffer before socket buffer allocation [20]. This positioning enables eBPF programs attached at the XDP code path to perform packet sanitization, filtering, and modification with minimal CPU overhead while keeping the integration with the kernel intact. This makes eBPF with XDP a viable framework for high-performance packet processing for tasks such as DDoS mitigation, ingress traffic shaping, and real-time intrusion detection and prevention [14] [32] [31].

### 1.5.1 Instruction limits and verifier evolution

The eBPF verifier enforces strict bounds on program size, stack usage and control-flow to guarantee in-kernel program termination and memory safety. Historically these limits were severe: pre-5.1 kernels constrained programs to maximum number of 4096 instructions which made non-trivial dataplane logic difficult and nearly impossible to express in a single eBPF program. From Linux 5.2 onwards further improvements allowed privileged users to load programs up to roughly 1000000 (one million) instructions in practice [18]. These kernel version dependent limits have important practical consequences for high performance packet processing designs. Even the relaxed instruction set allowance is orders of magnitude larger than before, it still requires a set of design patterns to be taken into consideration when building complex eBPF programs in order to stay within compliance range of the eBPF verifier. Some of the common design patterns are the following:

- Modularization via multiple small eBPF programs and `bpf_tail_call` dispatch so each program remains smaller, verifier-friendly and independently verifiable.
- Compact, pre-serialized data encodings that move parsing and complex data-layout work into user-space to reduce kernel instruction counts.
- Bounded and unrolled deterministically fixed size loops and fixed-size arrays iteration that allow the verifier to reason about loop bounds without introducing unbounded control-flow that can break memory safety requirements.

# 2 In-kernel packet filterer and sanitizer

This thesis introduces a simple, yet extensible system for in-kernel packet sanitization and normalization based on eBPF and XDP framework. High level system architecture overview is presented in Figure 1.

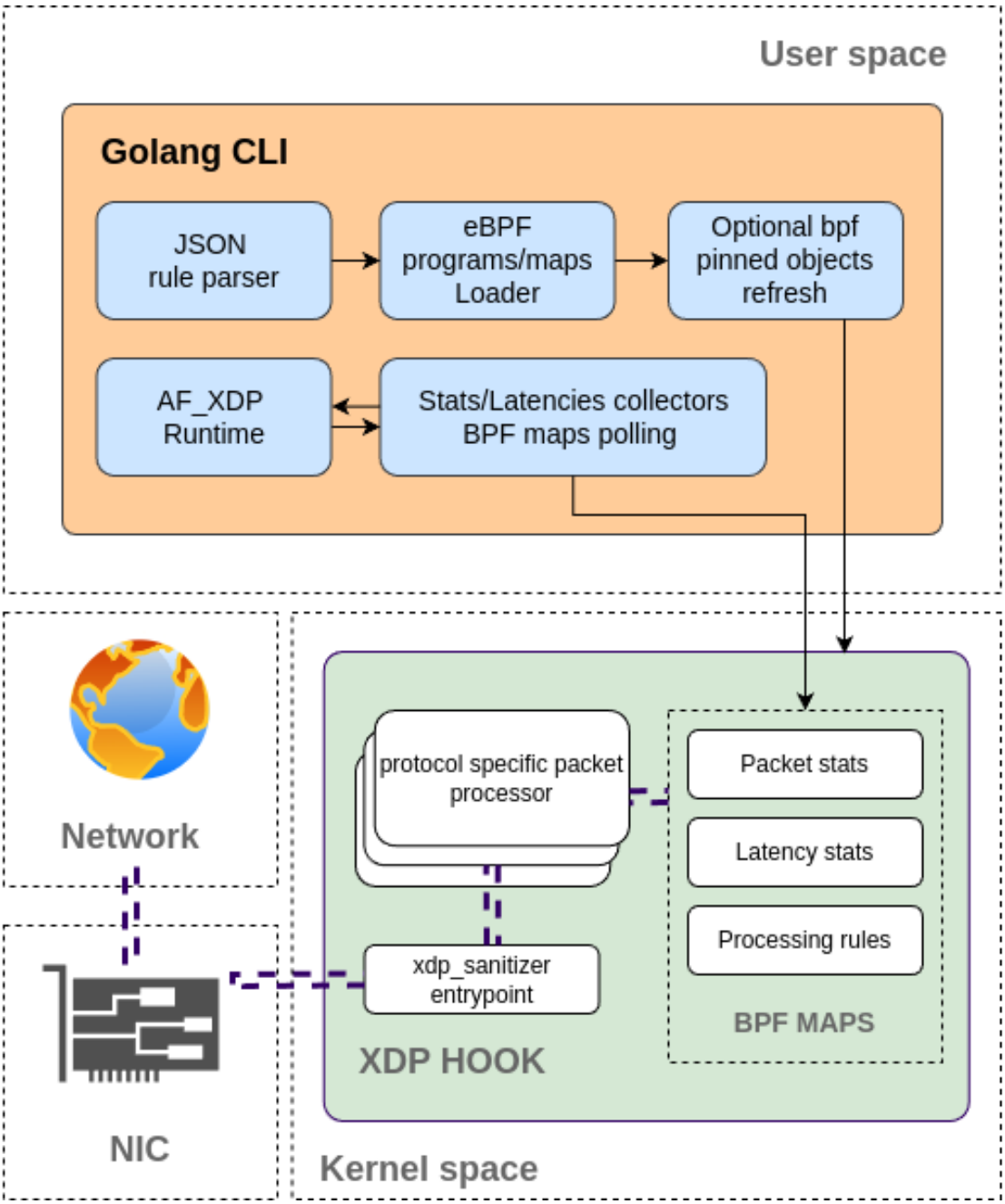


Figure 1. High level architecture overview

The user-space component is a command line interface application written in Golang. It is responsible for parsing user provided packet filtering and sanitization rules, generating and loading corresponding eBPF programs into the kernel and loading these provided packet processing rules into the kernel space eBPF programs. Go CLI application utilizes Cilium's eBPF library [4] that simplifies building, loading and managing of eBPF programs from within the user-space application. The tool ships with pre-built eBPF binaries suitable for running on both little and big endian architectures. However, the code for both userspace and kernel space is easily extendable for new

features. For new changes, the system requires recompilation of the user-space and kernel-space applications and must have llvm dependencies required for eBPF compilation installed (this can be achieved via containerization technologies such as Docker for better isolation and preventing environment pollution). Once eBPF C program code is generated, the Go CLI application invokes Cilium's eBPF library's bpf2go tool to compile the eBPF object files which are then loaded into the kernel.

The kernel-space bpf application utilizes BPF maps for storing rules, various statistics and other information need for efficiently performing cross boundary read and write operations from within the user-space CLI application. Any events that require further processing such as processed packet statistic counters, potentially complex packets that cannot be handled by the eBPF program or alerts are sent to the user-space application via these ring/array/per-cpu buffers and are processed within the user-space application.

System's kernel-space components are the generated and loaded eBPF programs, which manage packet processing at the XDP hook. These eBPF programs are attached to the XDP hook in the packet flow data path, which sits just right after the network interface card driver and before the Linux networking stack. This allows our system to process packets efficiently in-kernel before `sk_buff` allocation, without copying entire packet payload to user-space and requiring costly context switches. Additionally, processing packets in XDP hook in-kernel allows to avoid unnecessarily complex setups or polling NIC receive queue mechanisms found in user-space networking frameworks such as DPDK, thus reducing the CPU overhead. The system also supports running deep packet inspection to perform offloaded sanitization of complex packets which cannot be handled directly in eBPF programs.

The packet processing rules subsystem quietly powers the entire filtering and normalization system: the CLI reads in the user provided JSON rules file, translates them into TCP and UDP rule entries in user-space CLI program, writes them into dedicated eBPF maps at the load of eBPF programs - before anything touches the wire. When packets arrive to XDP hook, the main `xdp_sanitizer` program is always the first entrypoint which routes the packets to protocol specific packet processing eBPF program via bpf tail calls [11]. Protocol specific eBPF programs perform all the processing logic: consulting those rule maps, running rule predicates and sanitization and normalization methods, accounting for packet processing counters, and deciding which protocol-specific path should be taken for given packet. Because the rules state lives in bpf maps rather than compiled constants or eBPF code gen, small tweaks to sanitization behaviour travel from the CLI to the kernel with a simple reload, making the subsystem approachable even while it coordinates low-level primitives.

## 2.1 Packet filtering and sanitization rules system

Following the architectural overview of the overall packet filtering and sanitization pipeline, this subsection goes in depth and formalises the packet processing rules model that links declarative rules to network dataplane processing behaviour.

### 2.1.1 Setup Overview

The sanitizer consists of two tightly coordinated layers. The user-space control plane CLI tool, written in Go, and eBPF programs with bpf maps that bridges the user management plane and kernel space processing together. Userspace program processes and compiles user specified human-

readable packet processing policies into dense vectors of rules. These vectors are stored inside fixed-size eBPF array maps named `tcp_rules` and `udp_rules` for tcp and udp protocol packet processing respectively. Currently there is a hard limit and each map can hold at most 64 entries so that the eBPF verifier can unroll the rule evaluation loop safely without panicking or taking too much time during the eBPF program verification. Limit of 64 rules per protocol was chosen mostly as a rational and conservative enough value to comply with eBPF verifier and in-kernel program verification requirements and provide sufficient kernel-program loading speeds since each additional loop iteration increases the number of branches verifier has to explore which in turn increases the verification time and complexity. In the Golang user-space application side, every rule is encoded with the exact field order and alignment expected by the kernel side program, ensuring that the eBPF runtime can read it without additional parsing when loading rules from maps. In essence, user space application performs eBPF-C application binary interface compliant encoding of data which is stored in the BPF maps. Some specific fields such as tcp packet source or destination ports and their masks are encoded in network byte order to avoid endianness conversion in the kernel side which could quickly add up as unnecessary load when processing thousands of packets per second and applying dozens of rule predicated for each packet.

As mentioned previously, the in-kernel layer is realised by an eBPF program loaded in XDP hook that attaches to a network interface. The generic packet processing eBPF entrypoint called `xdp_sanitizer` performs inexpensive checks on the Ethernet and IPv4 headers and then dispatches the packet to a protocol-specific handler through a eBPF tail-call jump table. Two dedicated eBPF subprograms `packet_sanitizer_tcp` and `packet_sanitizer_udp` respectively are implemented for the TCP and UDP packet processing logic. Each handler iterates over the corresponding rule array from protocol rule map, applies the predicates to the live packet headers to perform a rule match, if matchd it executes any rule specific normalization steps, and finally returns an XDP or rule verdict (`XDP_PASS`, `XDP_DROP`, or `XDP_REDIRECT`). In case of `XDP_REDIRECT`, redirected packets are handed to an AF_XDP socket which would be capable of performing complex deep packet inspection. Drops halt further processing immediately, and pass verdicts allow the frame to continue through the kernel stack. Any headers data based packet normalization is done entirely in kernel space. Complex cases require AF_XDP socket offloading which can be performed directly within the network interface card receive queue buffers drastically improving the speed of processing or by copying to userspace if NIC does not support AF_XDP. This hybrid approach keeps rule policy processing flexible while keeping the fast path entirely inside the kernel.

### 2.1.2 Notation and Packet Model

The system operates directly on standard IPv4 and Transport Layer TCP and UDP packet headers extracted from raw incoming Ethernet frames. In the eBPF environment, these headers are accessed via direct memory pointers to the `struct iphdr` and `struct tcphdr` or `struct udphdr` definitions provided by the Linux kernel User API.

We model the packet state $\pi$ as a record of observable `tcphdr` TCP or `udphdr` UDP packet header fields. In the eBPF programs implementation the control plane serialises rules as match predicates into compact, C-ABI-compatible fields that the eBPF program evaluates with simple, usually integer comparison or bitwise operations. The fields relevant to matching are typed according to `tcphdr` or `udphdr` kernel C struct representations and serialized in network byte order when possible. Some of the possible rule match predicate fields are the following:

- **Src/Dst IP (CIDR):** Source and destination addresses that are matched as CIDR prefixes as `(RuleIp, RuleMask)` pair and serialized as 32-bit big-endian values (`__be32`). The kernel program performs masked comparisons (`((PacketIP & RuleMask) == (RuleIP & RuleMask))`). Masking allows to define a broad spectrum of matches and gives more flexibility than simple equality tests.
- **Src/Dst Port (masked):** Ports are matched using `(RulePort, RuleMask)` pairs and are stored as 16-bit big-endian values (`__be16`). The kernel eBPF programs evaluate masked ports in the same fashion as for IP addresses: `(PacketPort & RuleMask) == (RulePort & RuleMask)`. Masking allows to define a spectrum of matches and gives more flexibility than simple equality tests.
- **Flags:** TCP control flags are matched using an 8-bit value/mask pair (serialized as `__u8`). The kernel performs bitmask checks as folllowing: `(PacketFlags & RuleFlagMask) == (RuleFlags & RuleFlagMask)`.
- **Window:** The TCP packet window size is matched against an inclusive interval `[WindowMin, WindowMax]` (16-bit values, `__be16`). Or directly as single value, depending on the rule configuration.

### 2.1.3 Rule Definition

A packet processing rule $R$ is a structured object defining packet matching criteria, an action, and optional normalization parameters. The rule structure is designed to be memory-efficient and cache-friendly for the eBPF JIT compiler. Rules are a way for user to control the behaviour of the packet sanitizer by defining matching predicates and actions with verdicts. Rules are parsed in userspace program and loaded into eBPF maps for fast in-kernel evaluation.

$$R = \langle \text{MatchPredicate}, \text{VerdictAction}, \text{NormalizationSpec} \rangle$$

### 2.1.4 TCP Rules Definition and Implementation

A TCP processing rule $R$ is represented in the control plane by the Go type `TCPRule` and is serialized into the ABI-compliant C layout before being written into the eBPF `BPF_MAP_TYPE_ARRAY` map `tcp_rules` and utilized for TCP packet processing in the tail-called eBPF program named `tcp_processor`. The TCP rule tuples contains three main components mentioned in the previous generic rule definition:

- `MatchPredicate` predicate used to match a rule for a packet - CIDR, masked ports, flag value/mask, window size range;
- `VerdictAction` a matched rule action or a verdict - APPLY/PASS/DROP/REDIRECT (to AF_XDP) declared as `RuleAction` enum in Go;
- `NormalizationSpec` Normalization parameters encoded in a 16-bit bitmask `normalization_flags` which determines which normalization fields are enabled and should be performed during packet normalization step and a small set of auxiliary fields (for example `clamp_window_min/max`, `mss_clamp` for window size and maximums segment size adjustments respectively).

Enabled `MatchPredicate` rule predicate logic list is compacted into a single `__u32` field. Only predicates explicitly enabled by the rule's `fields_mask` are evaluated in the eBPF program. This design minimizes dataplane work, allows to simply skip disabled rules and not require having loops per rule and keeps the eBPF program verifier-friendly.

The defined set of matched rule verdicts

$$\mathcal{V} = \left\{ \begin{array}{l} \text{RuleActionApply}, \\ \text{RuleActionPass}, \\ \text{RuleActionDrop}, \\ \text{RuleActionOffloadAF\_XDP} \end{array} \right\}$$

map directly to XDP return codes:

- `RuleActionApply` → `XDP_PASS`
- `RuleActionPass` → `XDP_PASS`
- `RuleActionDrop` → `XDP_DROP`
- `RuleActionOffloadAF_XDP` → `XDP_REDIRECT`

The special `RuleActionApply` rule action indicates that matched packet should undergo normalization or sanitization by provided rule normalization operators. After normalization, packet is processed as `XDP_PASS` and sent up the kernel networking stack. `RuleActionPass` forwards the packet unchanged and issues the `XDP_PASS` return code. `RuleActionDrop` terminates processing immediately with `XDP_DROP` by dropping the packet, and `RuleActionOffloadAF_XDP` attempts to redirect the packet to an AF_XDP socket falling back to `XDP_DROP` if the redirect fails.

**Rule Match Criteria** A rule matches a packet $\pi$ if and only if **all** set of configured `fields_mask` match predicates return true. Any existing but not set fields are skipped. Table 1 details the available rule matching predicates and fields.

**Note:** The Go control-plane `Marshal` converts some of the high-level fields into the binary `tcp_rule` or `udp_rule` compliant data layout: `SrcCIDR`/`DstCIDR` becomes (addr, mask) pairs, ports and their masks are converted with `htons` to network byte order, and flags/masks are written as bytes. This pre-conversion avoids per-packet endianness conversions in the eBPF fast path.

**Rule Encoding and Serialization.** The Go control plane prepares a aligned binary descriptors `tcpRuleSpec` that mirrors the kernel side's `struct tcp_rule` rule representation.

- IP addresses and masks are stored as 32-bit big-endian integers via the helper that converts Go representation of IP `net.IPNet` into network byte order `uint32` values.
- Ports and port masks are converted with a host-to-network 16-bit `uint16` conversion in the control plane, so the kernel can compare packet fields directly without per-packet byte swaps.

| extbfField | **Type** | **Description** |
|---|---|---|
| SrcIP / DstIP | CIDR (RuleIp, RuleMask) | Masked IP match. Serialized as two 32-bit big-endian network byte order values (`__be32`) . |
| SrcPort / DstPort | Port (RulePort, RuleMask) | Masked port match. Serialized as 16-bit big-endian network byte order values (`__be16`); kernel checks `(PacketPort & RuleMask) == (RulePort & RuleMask)`. |
| Flags | Bitmap (`fields_mask`) (8-bit) | 8-bit flags matched by `(PacketFlags & RuleFlagMask) == (RuleFlags & RuleFlagMask)`. Serialized as `__u8`. |
| Window | Range [min, max] | Inclusive check: min $\leq$ window $\leq$ max. Serialized as 16-bit values (`__be16`). |

Table 1. TCP Rule Matching Predicates

- The `fields_mask` bitset indicates which predicate entries are valid for a given rule; when a predicate is unset the corresponding serialized values may be zeroed and are effectively ignored by the kernel rule matching loop.
- The `normalization_flags` 16-bit field encodes which rewriting operators (if any) should be applied when the rule matches.

Serializing rules in userspace shifts small, predictable CPU costs off the dataplane, reducing per-packet work and avoiding eBPF verifier-unfriendly dynamic branches.

### 2.1.5 Matching and Execution Semantics

The dataplane implements a deterministic, bounded policy: the kernel iterates over the fixed-size array `tcp_rules[0..MAX_TCP_RULES]` of rules where `MAX_TCP_RULES` defines the maximum upper bound of possible number of rules and sets this as static value of 64. Since the size of rules is known at compile time, the rule processing loop does not require annotations with a compile-time `#pragma unroll` since the fixed rules size allows the BPF verifier to reason about loop bounds and required instruction counts. This linear approach ensures deterministic behavior and is specifically designed to satisfy the eBPF verifier's constraints. The verifier requires all loops to be bounded and finite; a fixed array size of 64 allows the compiler to fully unroll the loop if necessary, guaranteeing termination without getting rejected when loading the eBPF program.

Generalized matching execution semantics look as follows:

1. Parse headers and dispatch to the protocol specific eBPF program handler via a tail-call.
2. For each enabled entry in `tcp_rules` or `udp_rules` (depending of the incoming packet protocol and bpf tail-call) evaluate the rules: test only the predicates selected by the rule's `fields_mask`. If currently evaluated rule does not match, continue to the next rule until all loaded rules are evaluated.

3. On the first rule that matches, apply normalization operators if there are any.

4. If a normalization operator returns a terminal signal the kernel program drops malformed/invalid packets in-place, the dataplane returns `XDP_DROP`.

5. Otherwise evaluate the rule action: `Drop` → `XDP_DROP`, `Pass` → `XDP_PASS`, `Offload` → attempt `AF_XDP` redirect (if redirect fails the implementation falls back to `XDP_DROP`).

6. Optionally, increment per-protocol statistics counters for matched/modified/dropped packets and latency traces.

The TCP packet processing eBPF program's `tcp_processing.c` rule evaluation logic for TCP protocol packets can be expressed in the following pseudocode in Algorithm 1.

**Algorithm 1.** TCP rule evaluation logic expressed in pseudocode

$\pi \leftarrow f$(Move ip packet pointer to tcphdr)
**for** $i = 0$ **to** $MAX\_TCP\_RULES$ **do**
  $R \leftarrow Rules[i]$
  **if** $R$.Enabled **and** Match($R, \pi$) **then**
    $\pi', NormAction \leftarrow$ ApplyNormalization($R$.Norm, $\pi$)
    **if** $NormAction ==$ XDP_DROP **then**
      **return** XDP_DROP
    **end if**
    **if** $R$.Action == Drop **then**
      **return** XDP_DROP
    **else if** $R$.Action == Offload **then**
      **return** XDP_REDIRECT
    **else**
      **return** XDP_PASS
    **end if**
  **end if**
**end for**
**return** XDP_PASS {Default verdict}

The implementation updates per-protocol statistics (match/modified/dropped counters) in a small per-CPU maps; these are useful for observability without impacting control flow decisions.

### 2.1.6 TCP Normalization Operators

Normalization operators are implemented directly in the kernel C code for each protocol tail call eBPF program and encoded as bit flags in `normalization_flags`. Each operator is implemented with careful bounds and verifier-friendly loops for example TCP options parsing is capped and unrolled. The main TCP protocol normalization operators implemented are the following:

- `SanitizeFlags`: rejects obviously invalid TCP flag combinations (e.g., SYN+FIN), returning a drop if the packet violates obvious state-machine rules.

- `ClearReserved`: zeroes reserved bits in the TCP header and updates checksums.

- `ClearECN`: clears ECE/CWR bits and updates checksums to avoid ECN-related anomalies.
- `FixUrgent`: synchronises the URG flag with the urgent pointer.
- `EnforceACK`: ensures that if the ACK flag is set, the acknowledgment number is non-zero, preventing certain evasion techniques.
- `ClampMSS`: when present, parses TCP options (bounded/unrolled) and lowers MSS to the configured `mss_clamp` to protect path MTU constraints.
- `EnforceWScale`: caps the advertised window-scale option to a safe `window_scale_max`.
- `ZeroTimestamps`: zeros timestamp option fields to avoid fingerprinting and wrap issues.
- `ClampWindow` and `ForceWindow`: clamp the receive window to a range or force a concrete value; checksum adjustments are applied atomically. The clamp range is sourced from the dedicated `clamp_window_min` / `clamp_window_max` fields, which must be present whenever `ClampWindow` is enabled so that normalization is independent of matcher-side window predicates.
- `StripSynData`: removes payload bytes from SYN packets, updates IP/TCP lengths and checksums, and adjusts the packet tail with `bpf_xdp_adjust_tail`. This operator is the most complex and is implemented with multiple safety checks (bounds, data_end comparisons) to preserve verifier stability.
- `ClearSynFin`: rewrites SYN+FIN anomalies into plain SYN by clearing FIN.

All rewriting operators update TCP/IP checksums using in-kernel incremental checksum helpers so that downstream network stacks see consistent packets.

### 2.1.7 UDP Processing

UDP rules are structurally similar but simpler: the serialized `struct udp_rule` contains fields for Src/Dst IP CIDRs, masked ports, an optional inclusive length range, and an optional checksum value/mask. The dataplane evaluates UDP rules in the same first-match, fixed-array manner as TCP, but no normalization operators are applied in-kernel for UDP-rules only instruct pass/drop/offload behavior. The code also validates UDP length against observed packet length before applying length predicates. Compared to TCP processing, UDP is simpler.

### 2.1.8 Safety and Design Rationale

The runtime behaviour and structure of the rule engine are chosen to balance performance, verifier safety, and expressiveness:

- Determinism and verifier-friendliness: fixed-size maps (64 entries) and bounded/unrolled loops let the BPF verifier statically reason about program termination and stack usage.
- Minimal per-packet work: the control plane serialises users provided rule values into network byte order whenever possible and writes a compact descriptor, so the kernel performs a few integer/bitwise comparisons rather than parsing or allocations.

- Precise predicates: using value/mask pairs and CIDR prefixes provides flexible, low-cost matching primitives that the kernel can evaluate with single machine instructions AND or COMPARE operations.
- Safe normalization: rewriting logic is only executed for matched rules and contains extensive bounds checks and incremental checksum updates to preserve packet validity and avoid memory-safety traps that would break the verifier.
- Observability: per-protocol counters (matched/modified/dropped) give control-plane operators visibility into sanitizer effects without changing packet processing semantics.

### 2.1.9 Extensibility Framework

The system architecture is explicitly designed to support future growth in three key dimensions: protocol support, rule complexity, and processing logic. This design ensures that the sanitizer can evolve to meet new security threats without requiring a complete redesign of the core dataplane.

**Modular Protocol Support.** The use of eBPF tail calls in the main `xdp_sanitizer` entry point decouples the protocol dispatch logic from the specific handlers. Adding support for a new protocol (e.g., ICMP or SCTP) requires only couple of steps:

1. Implementing a new eBPF subprogram (e.g., `packet_sanitizer_icmp`) conforming to the standard handler signature.
2. Updating the tail-call map in the control plane to route the new protocol number to the new program index.
3. Defining the rulesets and parsing logic in user-space program for the new protocol.

This allows the system to scale to new protocols without increasing the complexity or instruction count of the existing TCP/UDP paths and making sure the existing programs remain compliant with the strict eBPF verifier requirements.

**Normalization Extensibility.** The `normalization_flags` bitmask in the rule structure reserves space for future operators. Currently, 12 bits are used, leaving room for additional normalization fields within the 16-bit field. Adding a new normalization operator involves:

1. Defining a new bit constant in the shared Go/C header.
2. Implementing the transformation logic in the kernel C code.
3. Exposing the new option in the CLI configuration and control plane serialization.

Since operators are independent and composable, new security features can be added without disrupting existing logic.

**Flexible Rule Matching Criteria.** The rules `fields_mask` design allows the matching engine to be extended with new rule predicates matching other header fields (e.g., TTL, TOS, or TCP Options) or packet characteristics without breaking binary compatibility. The rule structure contains reserved padding bytes that can be repurposed for new field values, ensuring that the memory layout remains compatible while evolving.

## 2.2 Runtime architecture of the sanitizer

The entire packet sanitizer system can be described as three tightly coupled planes that are glued together through eBPF maps:

1. XDP hook entrypoint eBPF program `xdp_sanitizer` where every incoming packet of selected NIC arrives. This program does minimal packet header information parsing and basically acts as a packet router: decides which protocol handler should continue processing; It performs Ethernet/IP bounds checks, identifies a protocol number, and issues a bpf tail call via `bpf_tail_call` into the appropriate protocol processor stored inside the `protocol_programs` map (of a type `BPF_MAP_TYPE_PROG_ARRAY`).

2. Protocol specific processors such as `tcp_processor` or `udp_processor` that run the protocol specific rule matching predicates, applies packet normalization policies and performs the rule verdict on any matched packets. These programs are separate eBPF programs that live in a single loadable binary but in different ELF sections, and are therefore independently runnable and verifiable by the eBPF verifier.

3. A user-space Golang based CLI tool control plane that manages the lifecycle of eBPF programs, maps, parsing user provided sanitization rules and manages AF_XDP offload sockets.

There are few more non crucial but useful components in the sanitizer design. **Telemetry:** Lightweight counter bpf maps are stored in per-CPU arrays in each protocol specific tail-called bpf program (`tcp_stats`, `udp_stats`) so userspace program can aggregate packet matched/-modified/dropped numbers without introducing contention on the hot path. **Offload plumbing:** An optional dedicated bpf map `xsks_map` of type `BPF_MAP_TYPE_XSKMAP` that wires NIC receive queue directly to AF_XDP sockets created by the Go runtime. Offload or redirect decisions are encoded in the rule actions and executed in kernel context via `bpf_redirect_map`.

**AF_XDP Offload Runtime** The `AF_XDP` offload runtime enables a hybrid data plane where complex or resource-intensive packet processing is selectively redirected to userspace without the overhead of the standard kernel networking stack. When a rule triggers the `RuleActionOffloadAF_XDP` action, the eBPF program executes `bpf_redirect_map` against the `xsks_map` using the current RX queue index as the lookup key. The Go control plane manages this interaction by initializing `AF_XDP` sockets on startup, registering their file descriptors into the map, and maintaining background pollers to recycle descriptors and prevent ring exhaustion. This architecture allows for seamless integration of deep packet inspection or heavy logging pipelines while maintaining the safety and performance of the XDP fast path. Crucially, the system is designed to fail closed: if the userspace socket is unavailable or the ring is full, the kernel program detects the redirection failure and drops the packet to preserve overall system stability.

### 2.2.1 Entrypoint and tail-call fan-out

The `xdp_sanitizer` XDP entrypoint defined in `bpf/xdp_sanitizer.c` intentionally keeps its surface narrow: it validates the Ethernet and IPv4 headers, determines whether the payload is TCP or UDP packet, and performs a tail call into a bpf program entry stored in bpf map named `protocol_programs` which is a `BPF_MAP_TYPE_PROG_ARRAY` bpf map. Tail calls are preferable optimization to inlining in larger eBPF workloads because they allow the kernel eBPF

verifier to verify each protocol processor program individually. This is also important for extensibility of the system: if more handlers are added, the verifier complexity gets distributed across different verifiable units, rather than having a single large eBPF program. Tail calls also let the userspace loader swap implementations without reattaching the root program. When the requested program slot is empty, the kernel simply returns back to the entrypoint `packet_sanitizer`, which in turn returns `XDP_PASS` and allows the packet to continue through the normal networking stack.

There are currently 2 implemented protocol processors: one for TCP and one for UDP protocol packets. Both protocol processors follow the same structure and scaffolding: they re-parse Ethernet/IP headers and check any packet bounds to satisfy the verifier, compute protocol-specific packet metadata from the packet headers, and iterate through a fixed upper bound of `MAX_TCP_RULES` or `MAX_UDP_RULES` rules. The loops are statically bounded so that the verifier can statically prove bounded execution without rejecting the verification of each program. Each rule is a tightly packed struct stored in an array map as described in rules sections. Rule match fields are optional and flagged through bitmasks (`fields_mask`). TCP normalization goes further by editing headers in-place and recomputing checksums through helper wrappers such as `tcp_checksum_replace`. Edge cases such as SYN+FIN flag combinations, empty ACK numbers, or inconsistent URG state are handled explicitly via normalization rules, and any normalization modifications performed on the packet data adjust the buffers via `bpf_xdp_adjust_tail` kernel built-in helper.

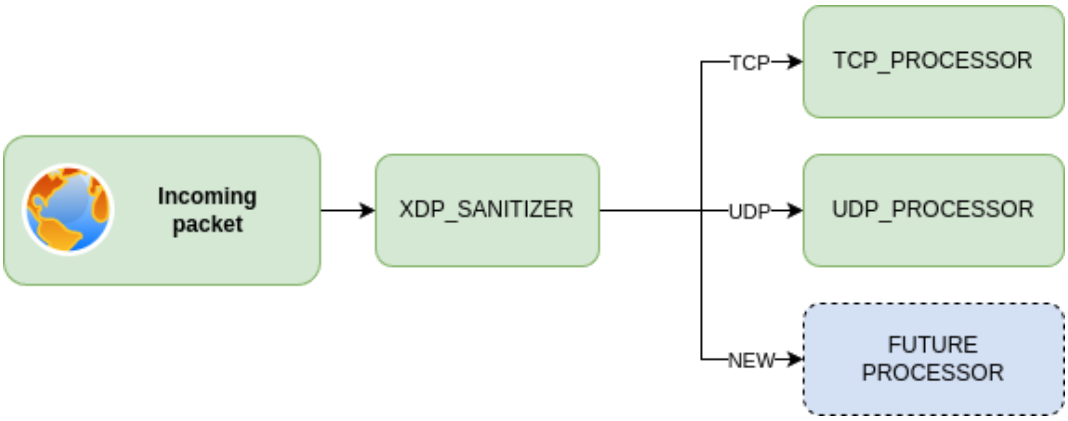


Figure 2. Incoming packet flow through the eBPF XDP chain of programs via tail-calls fan-out

### 2.2.2 BPF map primitives and pinning model

BPF maps act as the glue and communication bus between the kernel and userspace programs. Conceptually they are type-safe key/value stores that can be accessed by eBPF programs and user processes through file descriptors. The sanitizer uses a narrow set of map types: array maps for policy tables (`tcp_rules`, `udp_rules`), per-CPU arrays for statistics, the program array `protocol_programs` that stores file descriptors of tail-call targets, a ring buffer (`packet_events`), and an XSK map (`xsks_map`) that stores queue-to-socket associations. All of them are declared up-front in `maps.h` or the protocol source files so that `bpf2go` can autogenerate Go accessors which are used in the userspace program to simplify the management of reading and writing data to maps.

BPF map and program pinning allows to speed up the start-to-action time by persisting loaded objects by keeping maps and programs pinned inside the virtual bpf file sistem at `/sys/fs/bpf`.

The helpers in userspace program `internal/bpfutils/ebpf.go` implement a simple contract: `EnsurePinnedPrograms` and `EnsurePinnedMaps` try to reuse existing pins, fall back to pinning freshly loaded descriptors, and automatically delete stale entries that no longer load. This makes iterative runs faster and lets multiple processes share the same rule state by agreeing on a prefix. The CLI exposes `-pin-bpf` and `-unpin-bpf` so operators can explicitly transition between ephemeral and persistent deployments.

### 2.2.3 Userspace coordination

The Go entrypoint in `cmd/xdp/main.go` compiles all of the above into a coherent runtime flow. After parsing CLI flags and loading `rules.json`, it invokes the `bpf2go` generated loader, optionally swaps maps/programs with their pinned siblings, and writes protocol handlers into `protocol_programs` (keys 0 and 1 correspond to TCP and UDP). High-level rule structs defined in `internal/rules` are marshalled into the packed C representations and streamed into `tcp_rules` and `udp_rules`. Per-CPU statistics are periodically read through helper functions that sum counters across CPUs, allowing the CLI to print matched/modified/dropped figures without pausing the dataplane. If AF_XDP is enabled, the runtime in `internal/offload` opens a socket via `github.com/asavie/xdp`, registers it in `xsks_map`, and recycles descriptors in a background goroutine so that redirected packets never clog the queue.

### 2.2.4 Notable quirks and verifier-friendly techniques

Several small design decisions surface throughout the code base:

- **Verifier-friendly parsing:** Packet parsing is repeated in each tail call because the verifier treats programs as standalone entry points. This redundancy is intentional and keeps the verifier proof obligations simple.
- **Bounded option inspection:** TCP option parsing is capped at ten iterations, which covers the maximum option space yet lets the verifier unroll the loop. The parser records pointers to MSS, window scale, and timestamp fields so that normalization flags can operate without rescanning the header.
- **Rule density guardrails:** The rules arrays are fully populated on every start (unused slots are zeroed), which avoids leaking stale policies after a partial update and ensures deterministic iteration counts.
- **AF_XDP fallbacks:** When a redirect into `xsks_map` fails (for example because a userspace socket crashed), the program increments a drop counter and fails closed. This prevents resource exhaustion in the kernel queues.
- **Packet processing latencies:** The XDP entrypoint records start time of packet execution via `bpf_ktime_get_ns` at entry. Later in TCP/UDP processor programs the elapsed time is computed and recorded into latencies statistics which help evaluate the performance impact of different rule sets.

Together these mechanisms provide a modular sanitizer that is easy to reason about and extend. The control plane writes intent into maps, the XDP programs enforce that intent with predictable latency, and optional AF_XDP runtime lets selected flows be directed into userspace pipelines with all the hot path processing staying entirely in the kernel space eBPF programs.

## 2.3 Packet processing

The sanitizer's packet processing pipeline begins when network packets arrive at the XDP hook - just before the packet is passed into the kernel networking stack layers. Incoming packets are represented by the `xdp_md` struct containing metadata about the packet and pointers to the binary packet data. The eBPF program performs efficient packet inspection by directly accessing packet headers through pointer arithmetic on the raw packet data, validating packet boundaries to prevent buffer overflows that could crash the kernel.

To maintain modularity and avoid eBPF instruction limits, the eBPF programs are split into modular tail callable subunits that distribute packet processing based on the protocol as mentioned in Section 2.2.1. The entrypoint program `xdp_sanitizer` first parses the Ethernet header to confirm the packet protocol is IPv4 (`ETH_P_IP`). It then inspects the IP header's protocol field. If the protocol matches TCP (`IPPROTO_TCP`) or UDP (`IPPROTO_UDP`), the program invokes the `bpf_tail_call` helper function. This helper takes the context, a `protocol_programs` program array map, and an index to TCP or UDP processor program (`PROTOCOL_TAIL_TCP` or `PROTOCOL_TAIL_UDP`) to jump to the specific protocol handler program. This architecture allows for specialized processing logic for each protocol without bloating a single monolithic in-kernel eBPF program and enables the verifier to check each program independently.

As mentioned in previous Section 2.1.4 about rules. Each incoming packet in the tail-called program iterates through a configured set of rules stored in BPF maps (e.g. `tcp_rules` for TCP processor). This sequential evaluation checks packet fields against rule conditions such as IP addresses, ports, and flags. When a packet matches a rule's criteria, the corresponding action is executed as defined in Section 2.1.5. Once packet is processed in the sanitizer programs, it leaves into the next stages of the networking stack or is dropped at this stage whenever verdict is to drop it.

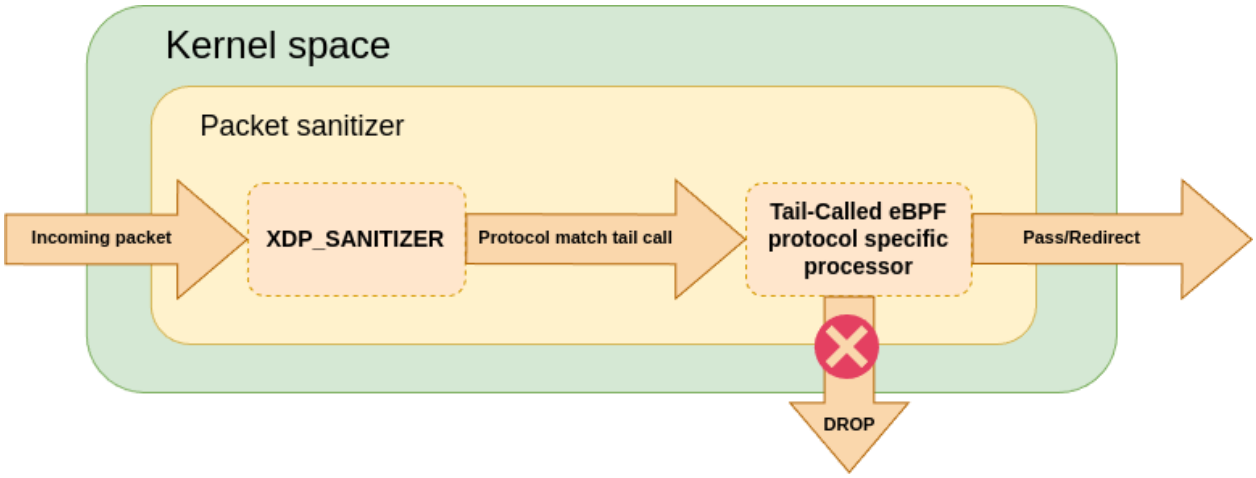


Figure 3. Packet processing flow in kernel and kernel programs

To provide visibility into the packet processing pipeline, the system maintains real-time statistics using per-CPU BPF maps (`BPF_MAP_TYPE_PERCPU_ARRAY`). Separate maps are allocated for TCP and UDP protocols (`tcp_stats` and `udp_stats`), each tracking four key metrics: total packets processed, packets matched by rules, packets modified by sanitization rule logic, and packets dropped. These counters are incremented atomically in the eBPF program, allowing user-space components to periodically poll and aggregate the data for monitoring traffic patterns and rule effectiveness without impacting the data plane performance. The packet processing system

also incorporates a high-performance per-packet processing latency tracking mechanism. Again, using per-CPU BPF maps (`BPF_MAP_TYPE_PERCPU_ARRAY`), the program records processing timestamps with minimal locking overhead. In order to minimize the overhead, only part of incoming traffic is sampled when collecting latency traces. This allows for pretty accurate performance monitoring of the packet processing pipeline, ensuring that latency sampling measures do not introduce unacceptable delays in the packet processing hot path.

## 2.4 JSON rule compilation pipeline

To decouple packet sanitization and processing rule authoring from eBPF bytecode generation, the user-space control plane accepts packet filtering and normalisation rules in a human-readable JSON format. The parser implemented in `internal/rules/json_parser.go` converts these declarative user provided rule descriptions from JSON format into strongly-typed Go structures that mirror the kernel struct layouts and later are passed into kernel program via bpf rules maps as defined in Section 2.1.4. Figure 4 summarizes rules parsing the flow: a rules file is streamed into the parser, protocol-specific builders validate each rule, and the resulting slices are pushed into BPF array maps via `LoadTCPRules` and `LoadUDPRules`. This configurable rule loading into the kernel program makes rule authoring convenient and simple. Operators of the tool can author rules without modifying anything in code and requiring recompilation of the project whenever something has to change on the policy side.

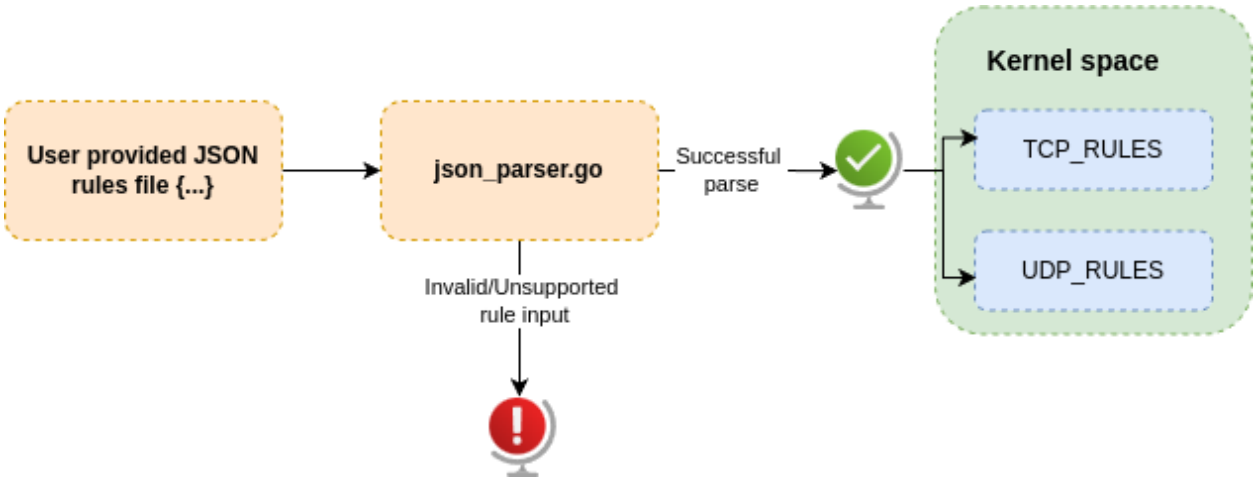


Figure 4. Simplified JSON rule file parsing and loading into BPF rule maps flow.

**Top-level schema.** The JSON document is parsed with Golang `encoding/json`'s streaming decoder to avoid unnecessary copies and to support large rule sets. Only a small set of keys is accepted: `version`, `metadata`, `tcp_rules`, and `udp_rules`. Version strings allow for future format migrations, while the optional metadata object can capture additional meta information data for example, automation job identifiers or commments or descriptions of rule file, without having any side effect for the dataplane.

**Protocol-specific rule decoding.** Each protocol rules array is delegated to a specialised parser function: `parseTCPRules` and `parseUDPRules` for TCP and UDP rules respectively. This design makes it easy to plug in additional new parsers for new protocols in future releases without touching existing logic. Every raw JSON rule is fed through `decodeStrictJSON`, which

enables `DisallowUnknownFields()` on the decoder. As a result, configuration errors such as misspelled field names or unsupported normalisation flags are surfaced with explicit error messages that include the rule index. The parsers also respects the BPF verifier's loop-unrolling limit by rejecting more than 64 TCP or UDP rules per array.

**Field validation and normalisation.** The builder helpers translate from loosely typed JSON into the strongly typed structures already used by the user-space CLI and kernel-space eBPF programs. Port matchers are expressed as optional value/mask pairs; when the mask is omitted, the parser defaults to an exact match. CIDR strings are parsed with the standard library's `net.ParseCIDR`, guaranteeing canonical network and mask encoding. TCP predicates may include flag masks and window ranges, while normalisation directives are specified as a string slice. The parser normalises these strings allowing case-insensitive, ignoring hyphens and underscores before mapping them onto the `TCPNorm` bitset. Any provided unknown normalization directive triggers a meaningful error, providing guidance for operators and CI pipelines alike.

**Rule action semantics.** Rule actions are supplied as strings, mapped to constants defined in `rules.RuleAction`. The current JSON parser implementation supports `apply`, `drop`, `pass`, and `offload_af_xdp action` fields which map to `RuleActionApply`, `RuleActionDrop`, `RuleActionPass`, and `RuleActionOffloadAF_XDP` actions respectively. Leaving the action blank defaults to `apply`.

**Extensibility of protocol rule parsers.** Two aspects make the parser future-proof. First, the protocol specific parser registry allows new protocols to be added without refactoring the control-flow of the top-level parser. Second, the helper functions isolate domain knowledge - for example, `buildWindowRange` checks `min <= max` and returns descriptive errors, so introducing new per-protocol fields is as simple as writing another helper function. The strict JSON handling also guarantees that any newly added field must be acknowledged on the configuration side, preventing silent ignores when requirements evolve. The CLI opens the rules file in streaming mode, so a missing or malformed configuration terminates startup of sanitizer with a clear error message. Combined with the JSON parser's strictness, this ensures the running eBPF program always reflects the operator's declared intent for filtering and sanitization rules. The output of `ParseRulesJSON` feeds directly into the existing `LoadTCPRules` and `LoadUDPRules` functions which feed the user provided and parsed rules into the eBPF program maps, so a new protocol would also require defining new `Load<PROTOCOL>Rules` functions to push the rules into the kernel program.

## 2.5 Synthetic packet generation framework

Due to the hardware limitations, synthetic packet generators were used to generate test traffic for testing the packet sanitizer system. Popular Python packet generation framework Scapy was used for this purpose. The packet generation system contains multiple scenario scripts in the form of `scapy_<name>.py` that exposes a function with the same name. The main packet-gen runnner entrypoint `run_generator.py` loads these scenarios, passes in provided network interface name and other scenario specific settings and executes the packet generation scenario on that local interface. During the testing, local machine used in generating synthetic packet payloads was peaking at roughly 300 thousand packets generated per second.

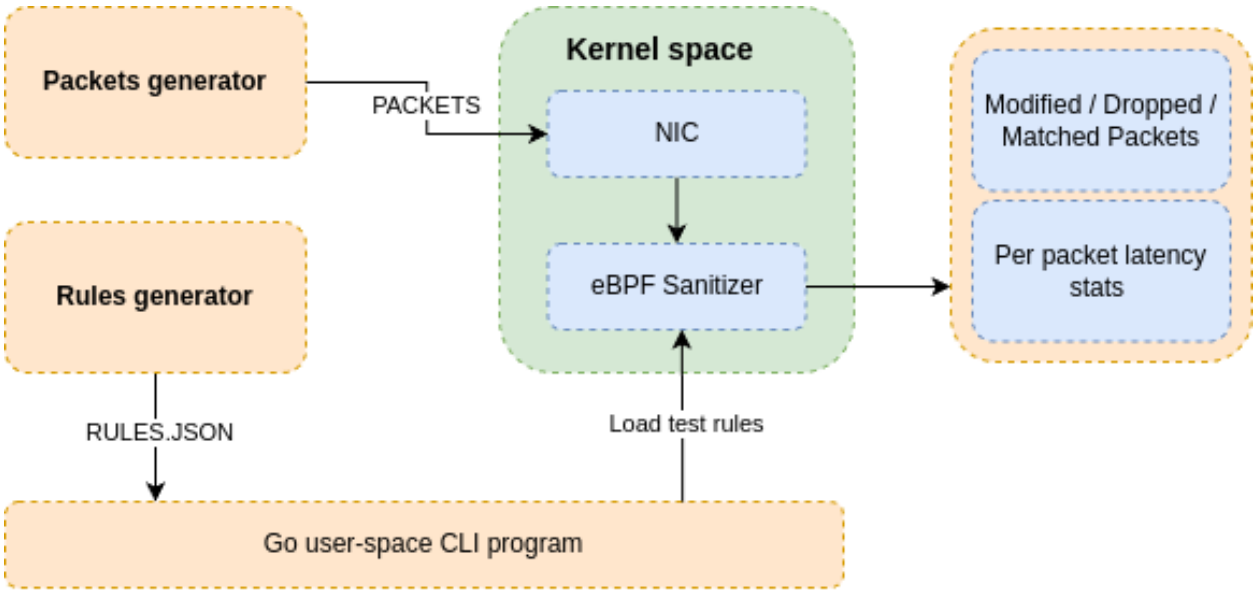


Figure 5. Synthetic packet generation framework and packet sanitizer stress testing flow.

**Packet sending with Scapy.** For sending packets we switch between two Scapy sending helpers. Simpler one `sendp` pushes frames straight into a `PF_PACKET` socket and is good for lower packet rates when we just want to inspect behaviour. More performant one `sendpfast` hands a batch of packets to `tcpreplay`, which replays the packets in native libpcap library code and can keep up with tens to hundreds of thousands of PPS generated on conventional hardware. Because of that dependency, `tcpreplay` and also `libpcap` has to be installed on the test host.

**Packet construction.** Every scenario starts from a canonical Ethernet/IPv4/TCP or UDP skeleton seeded with deterministic MACs and IPs of selected (`veth0`/`veth1` by default) NIC. TCP generators adjust sequence numbers, window sizes, and various options: for example, window scaling and SACK-permitted bits, destination and source IP addresses and ports. UDP flows toggle payload lengths and checksum masks. All this is done to make sure the generated packets match individual rule predicates.

**Veth workflow.** To get rid of any external internet noise, veth pairs are used as the test network interfaces. Every scenario targets are configurable, but by default all packet generators have their targets pointed to to the local `veth0`/`veth1` pair. The entrypoint packet generation runner can create and remove the pair via `-setup-veth` and `-teardown-veth` flags. Extra arguments passed after – reach the scenario directly, which lets us toggle PPS tiers or MAC addresses without editing code. Keeping these scripts alongside the sanitizer means anyone can reproduce the packet tests after cloning the repo. The full test loop is automated in a simple bash script: bringing up the veth pair via python scripts, start the packet sanitizer on `veth1`, blasting packets from via tcpreplay onto `veth0`, capturing stats, then cleaning up.

**Test rules.** List of various rule amount rulesets was generated from a master TCP rules ruleset using a Python script. Each rule file starting from the second includes all the previous rules from earlier rule file plus another new rule. From ruleset number 15 one udp rule is added into the rules list. Used rules excercise traffic patterns produced by the packet generator scenarios, ensuring that every synthetic packet matches a specific rule predicate. This tight coupling between

the rule generator and the packet generator allows for precise validation of the sanitizer's behavior, such as verifying that packets are correctly dropped, modified, or passed according to the loaded configuration or which rules have what impact on performance.

# 3 Results

## 3.1 Packet processing latencies

To evaluate efficiency and scalability of the eBPF packet sanitizer, we exercised the developed eBPF program with various distinct rulesets containing various amounts of different rules. In latency tests there were multiple rulesets varying from 1 to 18 rules loaded per rules file. Each loaded rule was specifically crafted to excercise a specific normalization or matching rule implemented in the eBPF kernel program. Synthetic packet generator was setup to produce multiple variable packet generation speeds starting from 1 thousand and going up to 500 thousand packets generated per second. In the testing system, almost all packet generation speeds met the expected PPS to be generated by the packet generator. Actual generated numbers were slightly lower for the largest PPS generation speeds above 300kpps since the test system was peaking at roughly 300 thousand packets (kpps) generated per second.

The packet traffic generator leans heavily on TCP because most of the implemented sanitizer actions target flag handling, window clamping, and payload trimming of the TCP protocol packets. Because of that, latency testing rule configuration files kept a healthy mix of both TCP and UDP rules and packets generated with larger skew towards TCP rules.

Python Scapy packet generation framework was utilized for generating packets used for stress testing the sanitizer. The python packet-gen script generates selectable number of packets to be generated per second: 1/10/50/75/100/150/200/500 thousand packets per second (kpps) that are sent to `veth0` virtual network interface card. Both the packet generator and the eBPF XDP packet sanitizer were running on the same machine performing the test, this migh have had some negligible performance impact for the overall processing latencies and results, but the impact was not measured due to the hardware constraints. In all of the tested flows, packets were produced and sent on `veth0` NIC which would deliver them to eBPF XDP sanitizer kernel program attached at the other end of veth pair at `veth1` NIC. At each test run, the sanitizer was allowed to run for 100 seconds for each combination of ruleset JSON file and kpps packet-gen rate, producing result trace files with JSON objects detailing the latency traces. Each output trace JSON document contains a newline delimited JSON objects ordered chronologically per each latency test run with a list of per-second latency stats along with multiple `slots` and `pps` fields for total number of matched/modified/-dropped packets in that second sample. Each slot contains latency trace information: number of samples counted, average per packet latency in ns, p95, and maximum latency. All latencies are counted in nanosecond scale. There are 3 main slots: global, TCP, UDP. Global slot traces all latencies stats for all types of packets from initial time the packet hits `xdp_sanitizer` until a `XDP` packet verdict is returned in a tail-called program. TCP slot traces TCP protocol packet processing time taken in tail-called eBPF program. UDP slot is analagous for TCP one, just for UDP packets. Additionally, there are per-rule slots that contain information of latencies for the loaded rules only and take into account the time from rule evaluation to rule verdict in a tail-called protocol processing program.

The kernel program periodically starts the sampling for an incoming packet, flushes the time taken to process the packet into a per-cpu bpf map along with the slot details such as protocol processing latencies, rule processing latencies. Downstream Golang user-space program collects the per-cpu map data every second, derives the average processing latencies and flushes the trace record JSON object into the trace output file.

Table 2 lists various different packet types described in the packet generator script. Each packet

type is designed to be targeted by a normalization rule that excercises the implementation of that specific normalization and sanitization logic in the eBPF XDP packet sanitizer kernel programs.

| **Generated packet name** | **Packet description** |
|---|---|
| API Gateway - SYN+FIN Probe | TCP probe from API Gateway with flags SYN, FIN enabled, window size set to 2048, no payload - exercises SYN+FIN normalization and clamping normalization. |
| Attacker - Mixed-Flags Probe | TCP probe from Attacker with flags SYN, FIN, RST, PSH enabled window size set to 1024, no payload - crafted anomalous flags to trigger drop/repair logic. |
| Frontend - SYN, FIN with Reserved Bits set | TCP probe from Frontend Web with flags SYN, FIN, window size set to 15000 and reserved bits set, no payload - tests reserved-bit sanitization together with odd flags. |
| Urgent Stream - PSH + UrgPtr | TCP from Urgent Stream with flags PSH, window size set to 64000, `urgptr` is set to 500, payload 1024 bytes - PSH with explicit urgent pointer and large window. |
| Urgent Stream - URG Flag Without Ptr | TCP from Urgent Stream with flags URG, PSH, window size set to 1501, payload 1024 bytes, but no `urgptr` set - verifies URG flag handling when urgptr is absent. |
| Frontend - MSS Option Packet | TCP from Frontend Web with flags ACK, window size set to 1500, TCP option MSS is set to 1460, payload 10 bytes - exercises MSS option parsing and small payload handling. |
| Frontend - Window-Scale Packet | TCP from Frontend Web with flags ACK, window size set to 65535, TCP option Window Scale is set to 14, payload 1200 bytes - tests large window handling with wscale. |
| IoT - Timestamp + WScale | TCP from IoT Control with flags ACK, PSH, window size set to 1200, options WScale and Timestamp is set to (12345,0), payload 50 bytes - tests timestamp and wscale normalization. |
| IoT - Large-Window Timestamp | TCP from IoT Control with flags ACK, PSH, window size set to 65535, same options, payload 50 bytes - large-window variant of timestamp+wscale test. |
| IoT - SYN with Options | TCP from IoT Control with flags SYN, window size set to 65535, timestamp and wscale options, payload 50 bytes - SYN carrying options for SYN-path normalization. |

Continued on next page

Table 2. List of packet types generated within scapy_3_latency.py packet generator Python Scapy framework for testing the average packet processing latencies in packet sanitizers

**Table 2 – continued from previous page**

| Generated packet name | Packet description |
|---|---|
| IoT - SYN+FIN with Options | TCP from IoT Control with flags SYN, FIN, window size set to 65535, timestamp and wscale options, no payload - SYN+FIN case including TCP options. |
| API Gateway - ACK (zero) | TCP from API Gateway 2 with flags ACK, window size set to 2048, ack is set to 0, payload 10 bytes - explicit ACK with zero acknowledgment number. |
| API Gateway - SYN with Data | TCP from API Gateway 2 with flags SYN, window size set to 2048, payload 1000 bytes - SYN packet carrying inline data to validate SYN-data handling. |
| API Gateway - ECN Flags | TCP from API Gateway 2 with ECN bits set, window size set to 2048, no payload - tests ECN/ECE/CWR sanitization. |
| DNS - Small Query | 64-byte UDP query from DNS Diag (53000→53), small DNS-like probe. |
| DNS - Large Query | 420-byte UDP query from DNS Diag (53000→53), large DNS-like payload to exercise drop paths. |
| NTP - Zero Checksum Probe | UDP from NTP Monitor (53100→123) with payload 76 bytes and UDP checksum set to 0 - validates checksum-guard logic. |
| Syslog - Medium Datagram | 160-byte UDP syslog datagram on port 1514 (src/dst) - medium-sized syslog traffic. |
| Telemetry - Jumbo Datagram | 900-byte UDP telemetry packet to Telemetry Jumbo (55000) - large UDP payload to stress jumbo handling. |

Table 2. List of packet types generated within scapy_3_latency.py packet generator Python Scapy framework for testing the average packet processing latencies in packet sanitizers

A total of 18 different packet filtering and sanitization rules were created in latency testing. Table 3 lists the rules and describes their functionality and what kind of normalization or filtering each rule performs. Each TCP rule is testing individual TCP processing normalization implementation. A broad range of rulesets was generated from a master ruleset JSON file containing all tcp rules. Each generated rules file contains rules that were in previous ruleset file plus a new one. After ruleset with 14 rules each subsequent ruleset file had additional one udp processing rule for lightwegith UDP packet matching/dropping.

| Rule name | Rule description |
|---|---|
| API Gateway - SYN+FIN Clamp | Matches API gateway SYN+FIN (src 10.10.2.20:40220 → dst 10.20.2.20:443). Applies ‘ClampWindow‘ to clamp TCP window to the configured range. Processes: API Gateway - SYN+FIN Probe. |
| Attacker - Invalid Flags (Sanitize/Drop) | Matches attacker flow (10.10.11.110:41110 → 10.20.11.110:666) with SYN+FIN and runs ‘SanitizeFlags‘ which drops obviously invalid flag combinations. Processes: Attacker - Mixed-Flags Probe. |
| Frontend - SYN+FIN (Clear Reserved Bits) | Matches frontend SYN+FIN (10.10.1.10:40100 → 10.20.1.10:80). Normalization ‘ClearReserved‘ zeros reserved TCP bits and updates checksums. Processes: Frontend - SYN+FIN (Reserved Bits). |
| Urgent Stream - PSH with UrgPtr | Matches urgent stream PSH packets (10.10.6.60:40660 → 10.20.6.60:8080) with large window; ‘FixUrgent‘ aligns URG flag and urg pointer. Processes: Urgent Stream - PSH + UrgPtr. |
| Urgent Stream - URG Flag Without Pointer | Matches urgent stream packets where URG flag present but pointer may be absent; ‘FixUrgent‘ clears or sets URG to keep state consistent. Processes: Urgent Stream - URG Flag Without Ptr. |
| Frontend - Clamp MSS Option | Matches frontend ACKs (10.10.1.10:40100 → 10.20.1.10:80). `ClampMSS` finds MSS option and reduces it to `mss_clamp` if larger, updating TCP checksum. Processes: Frontend - MSS Option Packet. |
| Frontend - Enforce Window Scale | Matches frontend ACKs with very large window; `EnforceWScale` caps the TCP window-scale option to `window_scale_max`. Processes: Frontend - Window-Scale Packet. |
| IoT - Zero TCP Timestamps | Matches IoT ACKs (10.10.3.30:40330 → 10.20.3.30:502). `ZeroTimestamps` zeroes TCP timestamp option fields. Processes: IoT - Timestamp + WScale. |
| IoT - Clamp Large Window | Matches IoT PSH+ACK with huge windows; `ClampWindow` clamps the window into configured bounds. Processes: IoT - Large-Window Timestamp. |
| IoT - Force Window Value | Matches IoT SYNs and forces the TCP window to a configured exact value (`ForceWindow`). Processes: IoT - SYN with Options. |
| | Continued on next page |

Table 3. Non exhaustive list of rules used in various rulesets for testing packet sanitizer packet processing latencies

**Table 3 – continued from previous page**

| Rule name | Description |
|---|---|
| IoT - Clear SYN+FIN Anomalies | Matches IoT SYN anomalies and `ClearSynFin` rewrites SYN+FIN into clean SYN (clears FIN). Processes: IoT - SYN+FIN with Options. |
| API Gateway 2 - Enforce ACK Number | Matches API gateway 2 ACKs (10.10.2.21:40220 → 10.20.2.21:443). `EnforceACK` ensures ACK packets have non-zero ack number (sets to 1 if zero). Processes: API Gateway - ACK (zero). |
| API Gateway 2 - Strip SYN Data | Matches API gateway 2 SYNs and `StripSynData` removes payload from SYN packets (adjusts IP/TCP lengths and checksums). Processes: API Gateway - SYN with Data. |
| API Gateway 2 - Clear ECN Bits | Matches API gateway 2 packets with ECN bits set; `ClearECN` strips ECE/CWR flags and updates checksum. Processes: API Gateway - ECN Flags. |
| Syslog - Sanitize | Matches syslog UDP (10.30.3.30:1514 → 10.40.3.30:1514); action `apply`/pass - intended to sanitize or accept syslog datagrams. Processes: Syslog - Medium Datagram. |
| Telemetry - Drop Jumbo Frames | Matches telemetry UDP (10.30.4.40:55000 → 10.40.4.40:55000) with length 800–1500 and action `drop` - drops jumbo telemetry frames. Processes: Telemetry - Jumbo Datagram. |
| DNS - Diagnostic Pass | Matches DNS diagnostic UDP (10.30.1.10:53000 → 10.40.1.10:53); action `pass` - allows diagnostic queries (matches both small and large DNS probes). Processes: DNS - Small Query; DNS - Large Query. |
| NTP - Monitor Pass | Matches NTP probes (10.30.2.20:53100 → 10.40.2.20:123); action `pass` - accepts NTP probes (including zero-checksum test packets). Processes: NTP - Zero Checksum Probe. |

Table 3. Non exhaustive list of rules used in various rulesets for testing packet sanitizer packet processing latencies

### 3.1.1 Mixed traffic global latencies

Figure 9 and Figure 10 presents a comprehensive view of the sustained global slot average packet processing latencies observed under mixed TCP and UDP traffic conditions across a wide range of ruleset sizes. Synthetic traffic was generating packets listed in Table 2 for each run. The collected packet processing latencies data demonstrates a stable and consistent patterns in processing time across varying packet generation rates, with processing latency trend lines for packet generation rates from 1 kpps to 500 kpps clustering tightly together per ruleset. The low average coefficient of variance howering below 1.2% per ruleset size indicates low fluctuations in packet processing latencies across different packet ingestion speeds. This stability indicates that the eBPF program's per-packet processing cost is largely independent of the traffic load, showing no signs of saturation or queuing delays even at higher throughputs and can be treated as a constant fixed size cost across different workloads. As seen in Table 4, Table 5 and Table 6 one distinction is observed that in low packet generation rates, especially at 1 kpps, the average latency measurements exhibit higher variability and higher overall latencies. This is explained by the fact that at lower pps rates, the sampling rate was also lower, therefore, higher rates move towards a more stable and consistent average latency measurement. Naturally, global slot contains latencies for all packet types and all rules, therefore, increase in ruleset size also increases the overall total average latency observed as seen in Figure 6.

| **Rules loaded** | **1 kpps** | **10 kpps** | **50 kpps** | **75 kpps** | **100 kpps** | **150 kpps** | **200 kpps** | **500 kpps** |
|---|---|---|---|---|---|---|---|---|
| **1** | 223.40 | 212.72 | 195.63 | 200.51 | 212.72 | 198.61 | 210.44 | 195.19 |
| **2** | 206.99 | 226.31 | 221.23 | 209.35 | 200.82 | 194.86 | 208.21 | 197.68 |
| **3** | 218.61 | 219.78 | 222.73 | 218.33 | 229.87 | 209.65 | 207.68 | 224.45 |
| **4** | 235.94 | 235.46 | 223.44 | 219.89 | 227.37 | 231.76 | 231.06 | 248.78 |
| **5** | 246.99 | 246.23 | 234.55 | 240.62 | 236.51 | 233.45 | 246.76 | 262.38 |
| **6** | 287.58 | 254.69 | 248.68 | 250.20 | 251.38 | 251.42 | 258.17 | 266.20 |
| **7** | 296.09 | 282.38 | 263.01 | 275.04 | 266.14 | 271.37 | 271.35 | 275.12 |
| **8** | 306.77 | 288.51 | 281.39 | 286.31 | 277.92 | 289.56 | 277.74 | 279.59 |
| **9** | 364.34 | 305.57 | 292.63 | 294.10 | 290.26 | 303.12 | 304.21 | 290.00 |
| **10** | 330.73 | 317.52 | 307.66 | 310.43 | 316.60 | 316.71 | 336.18 | 312.20 |
| **11** | 345.09 | 311.64 | 304.02 | 315.01 | 318.64 | 306.19 | 307.71 | 304.08 |
| **12** | 350.37 | 327.11 | 321.59 | 322.97 | 329.47 | 368.53 | 327.67 | 349.90 |
| **13** | 402.12 | 361.55 | 354.15 | 352.08 | 355.81 | 351.29 | 371.32 | 371.09 |
| **14** | 389.24 | 377.94 | 363.69 | 370.34 | 371.57 | 372.85 | 364.46 | 395.27 |
| **15** | 397.97 | 380.33 | 385.72 | 377.63 | 376.50 | 376.29 | 391.83 | 390.15 |
| **16** | 417.95 | 388.74 | 386.31 | 388.06 | 390.78 | 378.99 | 384.46 | 383.69 |
| **17** | 391.57 | 384.98 | 379.80 | 377.04 | 381.82 | 380.23 | 379.77 | 376.76 |
| **18** | 400.49 | 386.60 | 373.51 | 373.73 | 383.12 | 379.63 | 376.73 | 375.37 |

Table 4. Average global slot packet processing latencies (ns) across tested ruleset sizes and packet generation speeds. First column indicates the number of rules loaded from the configuration file, while the top row indicates the packet generation speed in kilo packets per second (kpps).

Decomposing this global slot latency metrics into its protocol components reveals further insights. Figure 8 and Figure 7 depicts combined UDP and TCP packet processing average latencies per loaded ruleset. TCP packets processing slot `tcp_total` average latencies depicted in Figure 7 tightly resemble the `global` slot latencies up until 15 rules loaded. From 15th ruleset onwards we start to load UDP rules into the eBPF XDP packet sanitizer and see a slight upward movement in

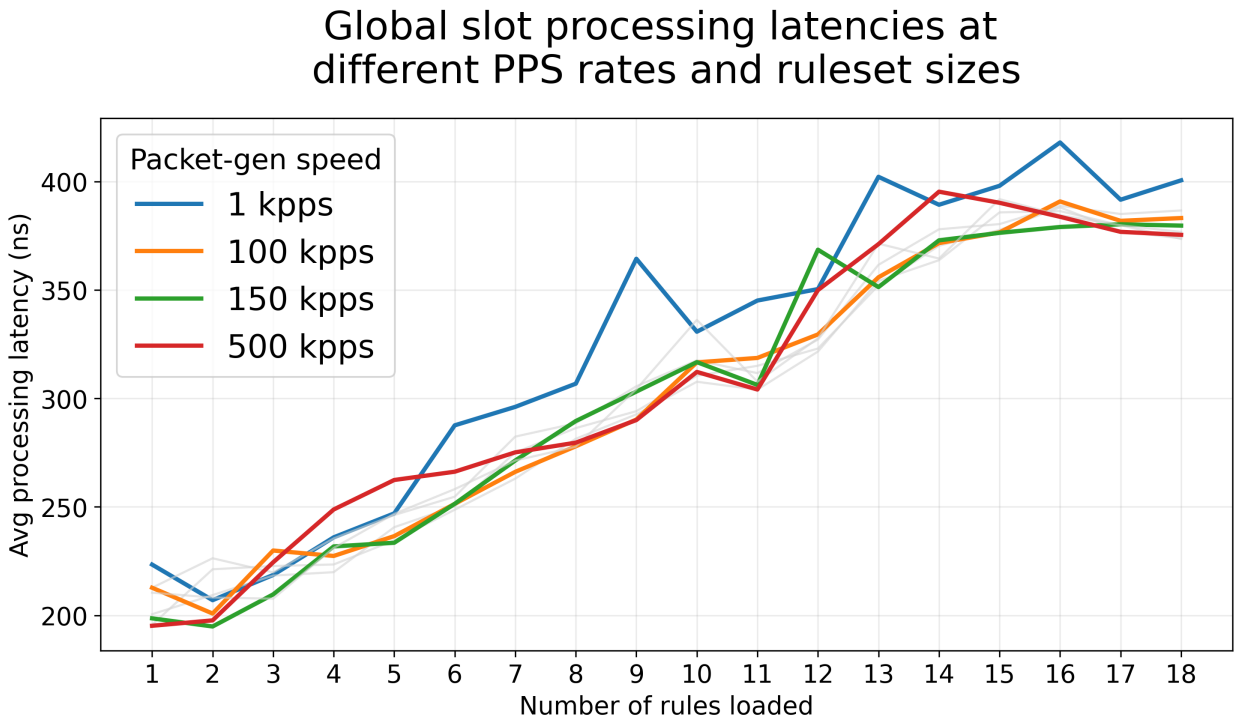


Figure 6. Average global slot packet processing latencies at different tested ruleset configuration file rule amounts and synthetic packet generator speeds. Outlined 4 representative packet generation speeds, other speed lines are grayed out. See Tables 2 and 3 for packet types and rules used.

UDP latencies as rules start to kick in action. Since UDP rules are simpler than TCP rules in terms of complexity and computational cost, we see the notional average processing latencies for UDP are smaller than TCP. The introduction of UDP rules into the test also drags the overall global slot average latencies down a bit and TCP latencies maintain their baseline reached at 14 rules.

Nevertheless Figure 6, Figure 7 and Figure 8 shows a clear overlap of average packet processing latencies at the points of same ruleset size and packet processing speeds confirming the robustness and evidence of constant performance per ruleset without large drifts or resource exhaustion over time. A small discrepancies at lower 1kpps packet generation rate is insignificant given the overall latency sampling size was much smaller at that packet generation speed.

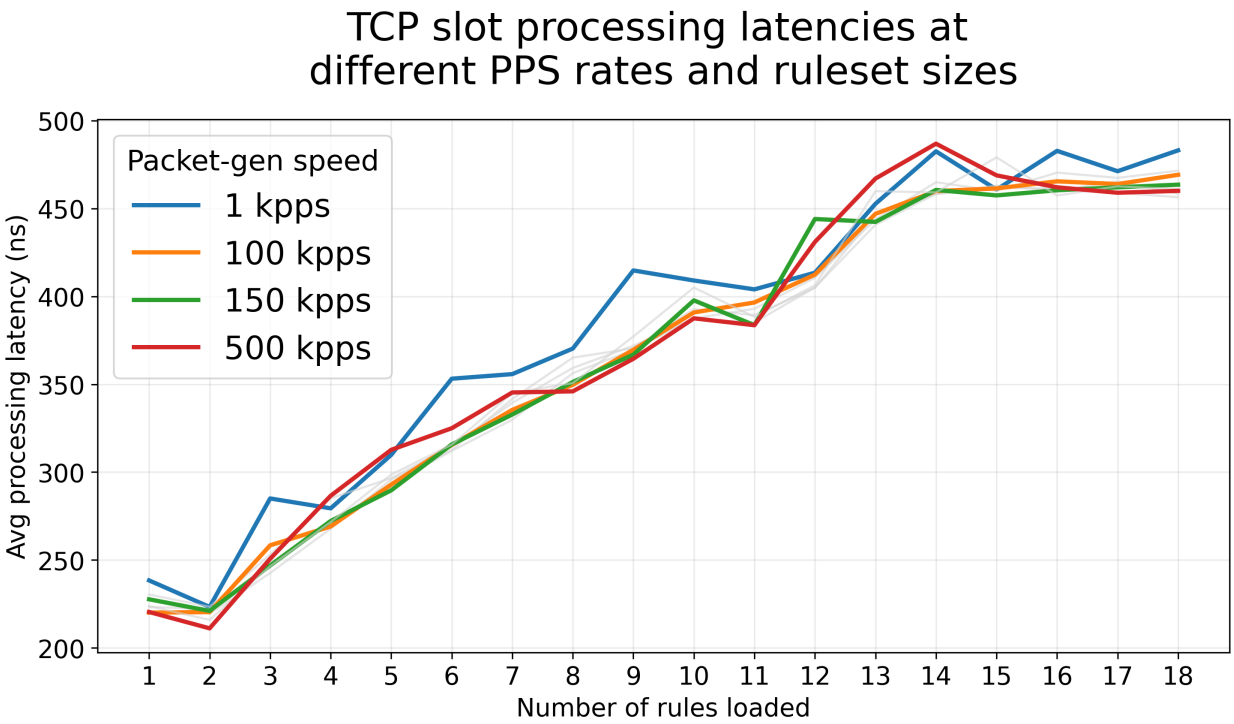


Figure 7. Average tcp slot packet processing latencies at different tested ruleset configuration file rule amounts and synthetic packet generator speeds. Outlined 4 representative packet generation speeds, other speed lines are grayed out. See Tables 2 and 3 for packet types and rules used.

| **Rules loaded** | **1 kpps** | **10 kpps** | **50 kpps** | **75 kpps** | **100 kpps** | **150 kpps** | **200 kpps** | **500 kpps** |
|---|---|---|---|---|---|---|---|---|
| **1** | 238.38 | 230.37 | 223.34 | 227.69 | 220.06 | 227.64 | 223.71 | 220.43 |
| **2** | 223.28 | 223.35 | 222.02 | 219.34 | 220.36 | 221.04 | 215.92 | 211.11 |
| **3** | 285.01 | 253.03 | 248.28 | 242.60 | 258.35 | 246.85 | 246.44 | 250.69 |
| **4** | 279.40 | 285.16 | 271.30 | 267.85 | 269.08 | 272.21 | 271.61 | 286.54 |
| **5** | 309.82 | 296.56 | 293.35 | 295.77 | 292.92 | 289.63 | 298.62 | 312.68 |
| **6** | 353.15 | 314.78 | 312.10 | 312.57 | 315.44 | 315.75 | 315.64 | 324.97 |
| **7** | 355.76 | 339.42 | 329.85 | 341.64 | 335.41 | 332.76 | 345.14 | 345.32 |
| **8** | 370.30 | 359.29 | 356.34 | 365.14 | 349.67 | 351.30 | 350.66 | 345.96 |
| **9** | 414.76 | 371.89 | 369.59 | 370.47 | 369.62 | 366.84 | 377.16 | 364.48 |
| **10** | 409.03 | 387.45 | 387.23 | 392.79 | 390.84 | 397.71 | 405.04 | 387.51 |
| **11** | 403.99 | 392.95 | 385.11 | 389.34 | 396.52 | 383.65 | 388.33 | 383.60 |
| **12** | 413.46 | 410.57 | 405.11 | 404.82 | 412.40 | 444.02 | 406.30 | 430.98 |
| **13** | 452.67 | 443.81 | 440.75 | 446.80 | 446.96 | 442.39 | 459.85 | 467.03 |
| **14** | 482.54 | 464.95 | 458.39 | 458.13 | 459.89 | 460.56 | 459.18 | 486.89 |
| **15** | 460.77 | 460.03 | 463.03 | 462.00 | 461.43 | 457.45 | 479.10 | 468.85 |
| **16** | 482.79 | 470.42 | 460.19 | 461.18 | 465.46 | 460.38 | 457.38 | 462.05 |
| **17** | 471.22 | 467.42 | 465.46 | 458.95 | 463.91 | 462.05 | 462.62 | 458.97 |
| **18** | 483.08 | 471.52 | 458.50 | 456.42 | 469.18 | 463.56 | 461.01 | 460.05 |

Table 5. Average TCP slot packet processing latencies (ns) across tested ruleset sizes and packet generation speeds. First column indicates the number of rules loaded from the configuration file, while the top row indicates the packet generation speed in kilo packets per second (kpps).

| Rules loaded | 1 kpps | 10 kpps | 50 kpps | 75 kpps | 100 kpps | 150 kpps | 200 kpps | 500 kpps |
|---|---|---|---|---|---|---|---|---|
| **15** | 263.27 | 229.58 | 241.39 | 220.50 | 219.59 | 228.14 | 231.54 | 245.30 |
| **16** | 279.70 | 238.03 | 249.21 | 252.82 | 251.99 | 230.09 | 250.26 | 239.05 |
| **17** | 241.35 | 233.77 | 222.86 | 224.39 | 230.05 | 229.50 | 226.61 | 225.06 |
| **18** | 253.19 | 229.37 | 219.74 | 220.97 | 226.73 | 225.33 | 222.25 | 219.29 |

Table 6. Average UDP slot packet processing latencies (ns) across tested ruleset sizes and packet generation speeds. First column indicates the number of rules loaded from the configuration file, while the top row indicates the packet generation speed in kilo packets per second (kpps).

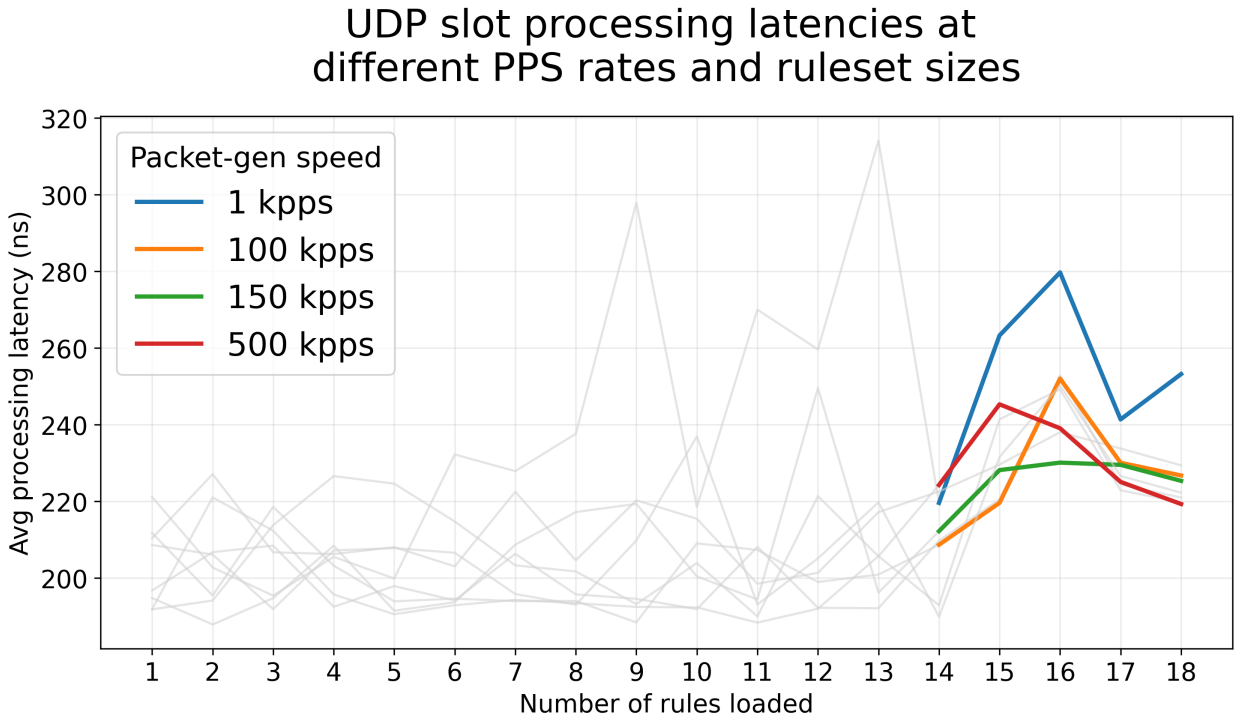


Figure 8. Average UDP slot packet processing latencies at different tested ruleset configuration file rule amounts and synthetic packet generator speeds. Outlined 4 representative packet generation speeds, other speed lines are grayed out. Since UDP rules are utilized from ruleset with 15 rules onwards, the plot highlights data from that ruleset size. See Tables 2 and 3 for packet types and rules used.

Average global slot packet processing latencies sustained over time for different rule sizes and packet-gen speeds (mixed TCP and UDP traffic) - Part 1/2

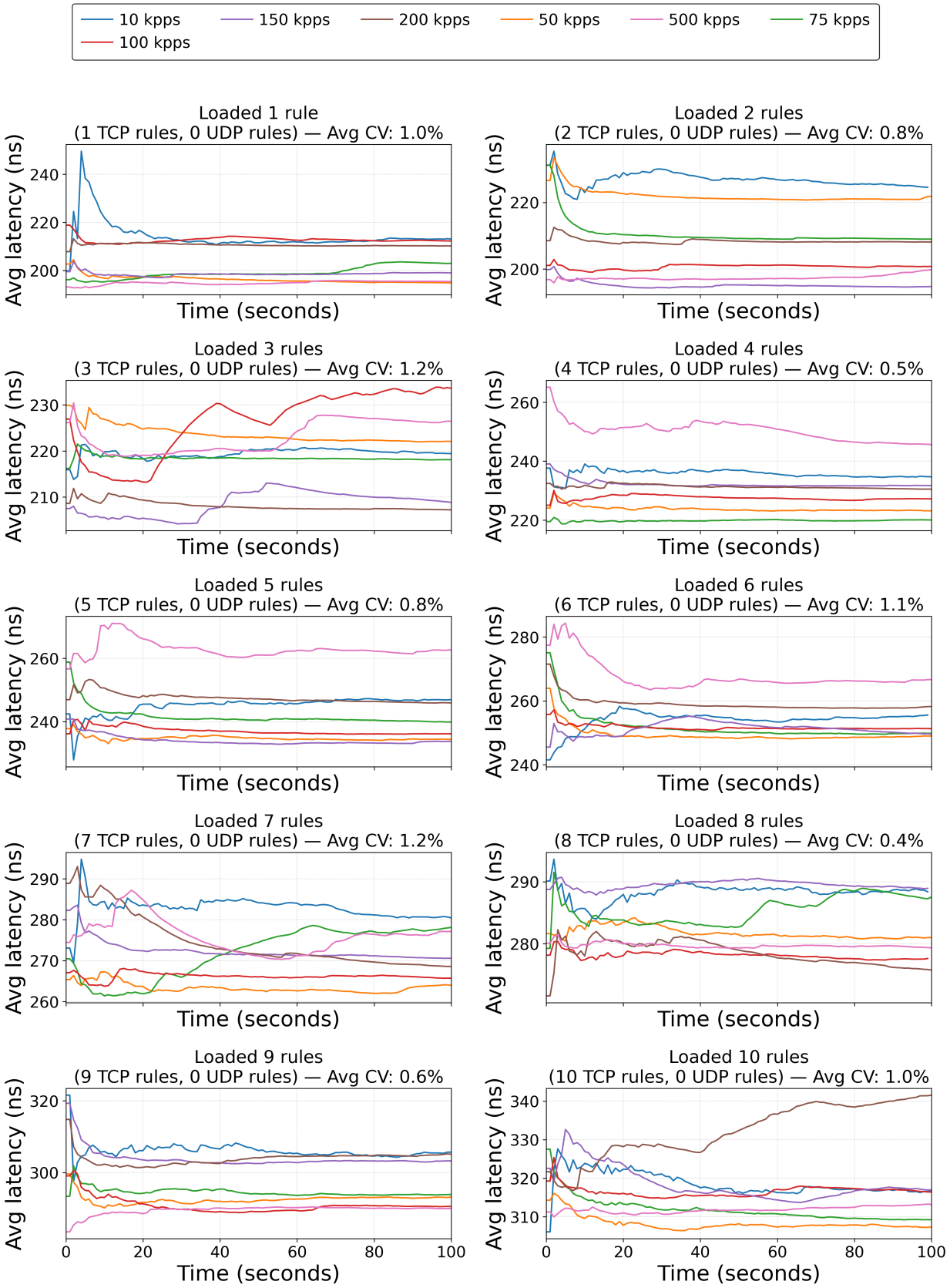


Figure 9. Global slot average packet processing latencies sustained in time across the tested rule-sets and generated packet speeds for mixed TCP and UDP traffic. Avg CV denotes the averages coefficient of variances across all packet-gen speeds per ruleset. (1/2)

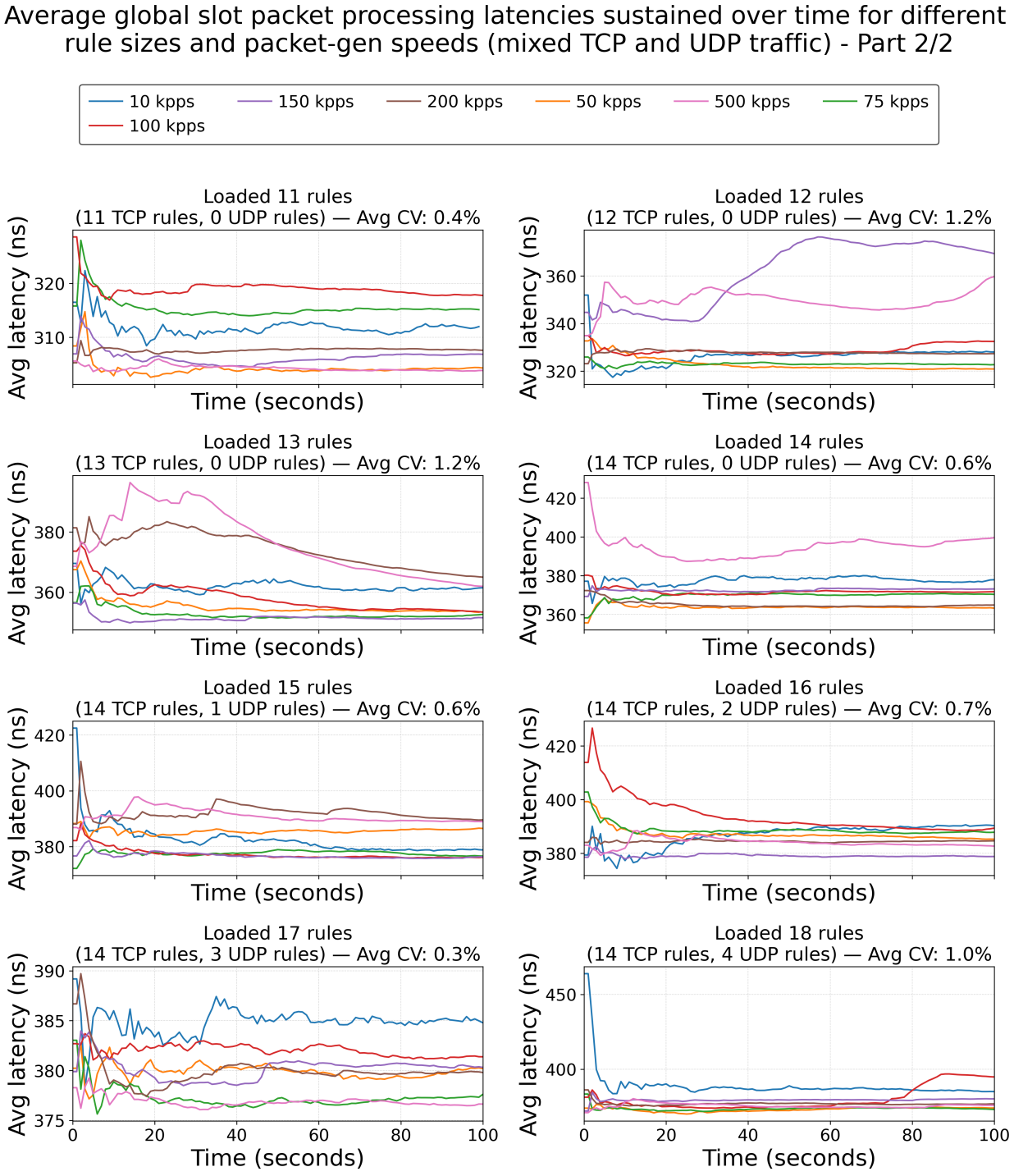


Figure 10. Global slot average packet processing latencies sustained in time across the tested rule-sets and generated packet speeds for mixed TCP and UDP traffic. Avg CV denotes the averages coefficient of variances across all packet-gen speeds per ruleset. (1/2)

Figure 13 confirms that even at high packet processing speeds, the average per-rule processing latency stays consistend over time and can be treated as a constant fixed cost per packet processed by that rule. The only factor in packet processing latency is the complexity of the rule itself: computational cost and memory writes incurred by the packet modification logic. Different packet generation and processing speeds do not have a any significant impact for the average latencies observed. Figure 11 and Figure 12 depict heatmap charts of observed averages for processing latencies for each TCP and UDP rule per each packet generation speed. We can see that the latencies remain stable across different packet generation speeds with no significant deviations across different pps rates.

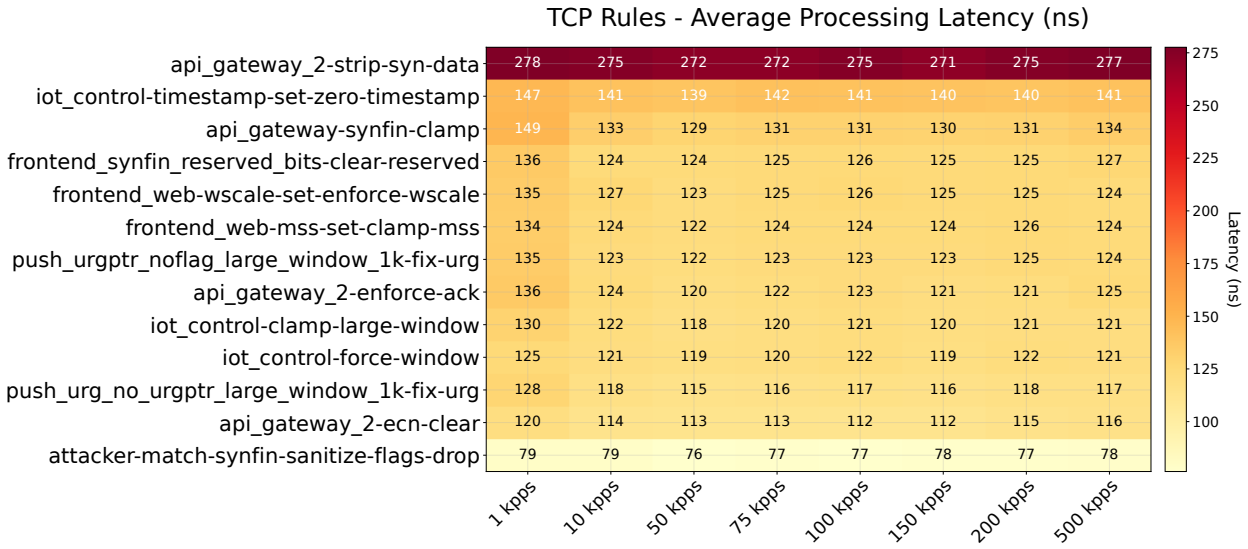


Figure 11. TCP rules latency heatmap across different packet generation speeds sustained over 100 seconds runs

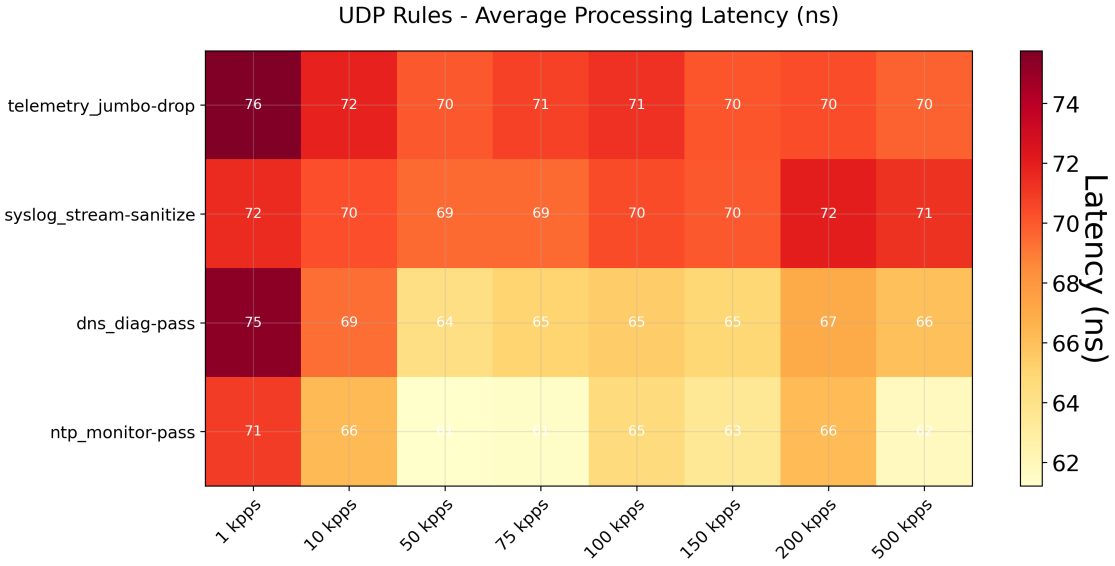


Figure 12. UDP rules latency heatmap across different packet generation speeds sustained over 100 seconds runs

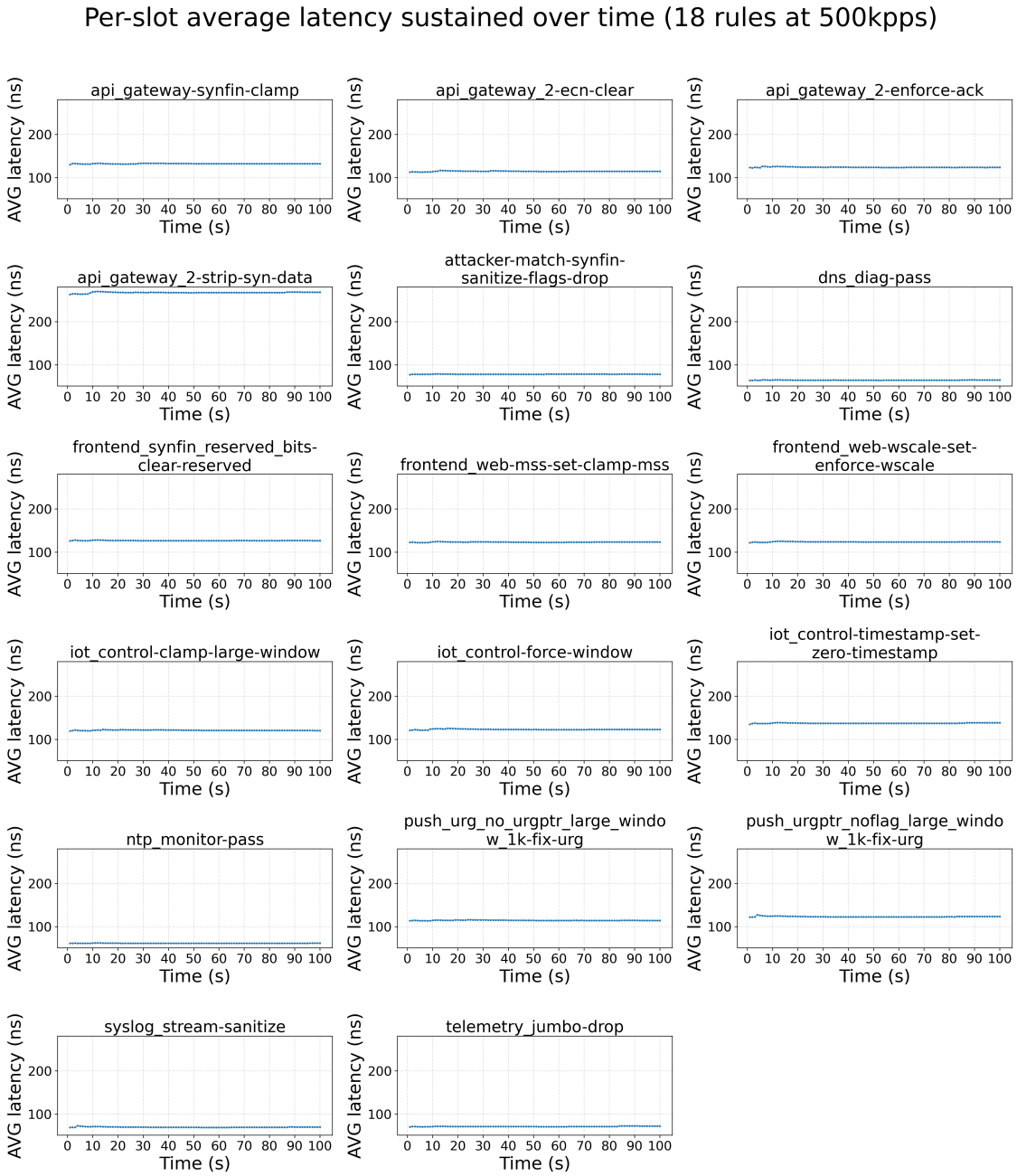


Figure 13. Average packet processing latencies per rule sustained for 100 seconds with 18 rules loaded for mixed TCP and UDP traffic. Each subplot depicts packet processing latencies for a given rule.

Overall, packet processing latencies under mixed TCP and UDP traffic conditions demonstrate consistent and predictable performance characteristics. The eBPF XDP sanitizer is efficient in handling diverse traffic and multitude of rulesets at the packet ingestion rates on conventional hardware without significant performance degradation.

## 3.2 Comparison of eBPF/XDP and Netfilter's NFQUEUE

Proof-of-concept program for performing simple packet sanitization was developed using Netfilter's NFQUEUE subsystem in order to compare alternative system's performance to the developed in-kernel eBPF/XDP sanitizer. The developed NFQUEUE program would perform a simple action: capture incoming TCP packets and inspect the TCP header `window-size` field. For simplicity and for the sake of performance evaluation, window scaling was not taken into account, just the pure window size value. All packets with window sizes larger than 65000 bytes were sanitized by setting the window size value to 65000 bytes. This program is functionally mimicking the `ForceWindow` TCP packet normalization rule implemented in the eBPF/XDP sanitizer.

The Netfilter NFQUEUE user-space offloading program was implemented in Golang using the `go-nfqueue` package [16]. The program sets up netfilter rules using the `iptables` utility to capture all incoming TCP traffic on a given interface and offload the packet processing to the user-space NFQUEUE program - in this case the Go program. The Go program establishes communication with kernel-space via an `AF_NETLINK` socket that is created during the registration of the NFQUEUE handler. Once running, the NFQUEUE program receives the incoming packets for the selected NIC, performs the packet inspection and sanitization operation: inspects TCP packet headers, modifies the TCP window size, and prints the number of processed and modified packets per second.

The XDP/eBPF packet sanitizer had 1 rule loaded that performs the same operationanl logic as the NFQUEUE one. Rule would match incoming `iot_control-large-window-syn` (see Table 2) packets and sanitize the TCP window size field by performing `ForceWindow` normalization action. Both programs were let to run for 100 seconds while the veth0/veth1 virtual NIC was being bombarded with synthetic traffic of around 500 (up to 300 on the tested hardware) thousand packets generated per second.

Testing was performed on a machine with the following specifications:

- AMD Ryzen 5 PRO 5650U processor (12 cores)
- 8 GB DDR4 RAM
- Linux kernel 6.8.0-47-generic
- `/proc/sys/net/core/bpf_jit_enable` set to 1 for eBPF JIT compilation
- XDP mode: generic (without hardware offloading)

Figure 14 shows the comparison of packet processing capabilities for both created programs. It is clear that both implementations sustain their reached baseline without major deviations. NFQUEUE program manages to process roughly 43 thousand packets per second, while the eBPF/XDP one stays consistent at around 220 thousand packets processed per second. We can clearly see the difference between both implementations is almost fice-fold in favor of the XDP/eBPF implementation. These results validate the claim that eBPF programs at the XDP hook in the network data path can substantially outperform generic user-space offloading mechanisms like Netfilter's NFQUEUE.

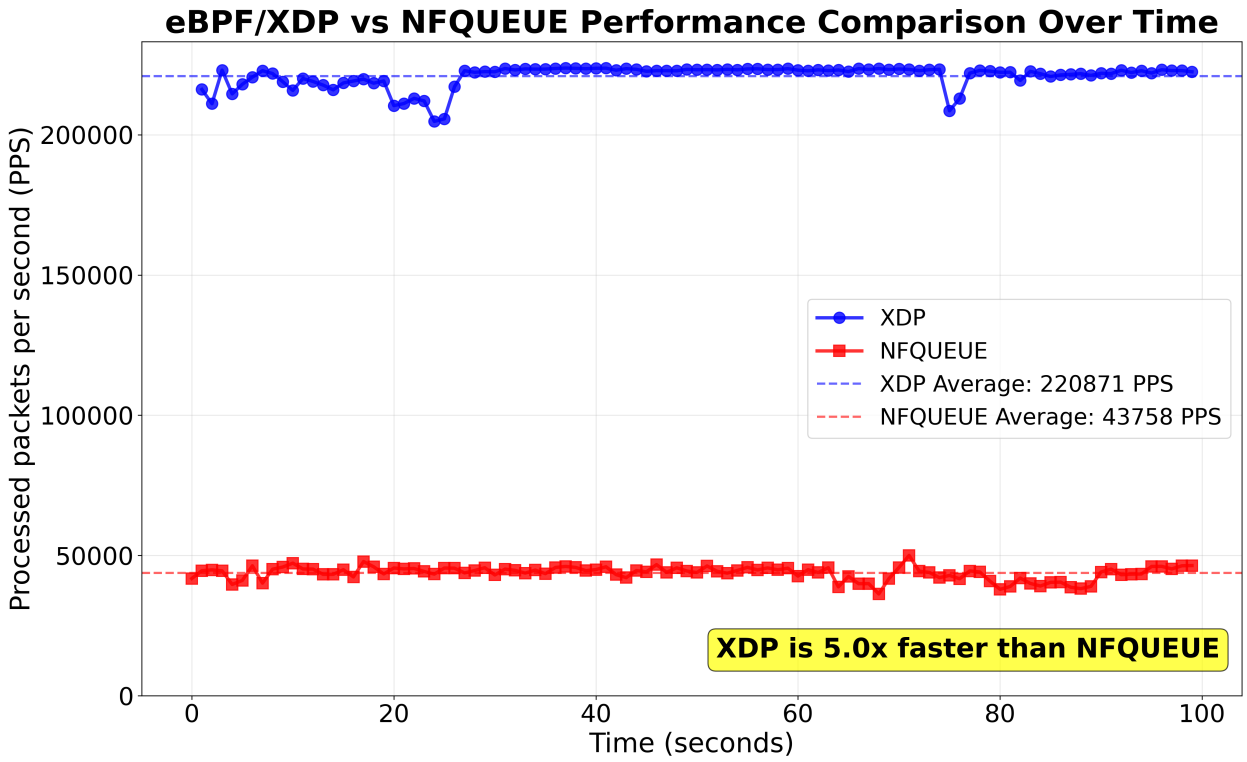


Figure 14. Comparison of packet processing performance for a simple TCP packet window size sanitization rule between eBPF/XDP packet sanitizer and reference netfilter's NFQUEUE implementation.

# Conclusion

This thesis presents a practical design and implementation overview of an in-kernel packet sanitizer system based on the eBPF/XDP framework with a Golang user-space control plane. The work demonstrates that a eBPF verifier friendly packet processing dataplane implemented in kernel-space using eBPF and XDP framework, leveraging eBPF maps, pre-defined and pre-serialized packet processing rule descriptors, bounded loops, and bpf tail-calls can perform a deterministic and low-cost transport layer protocols packet processing and sanitization without compromising on performance and extensibility.

The empirical performance evaluation of implementation shows that per-rule processing cost is mostly bounded and predictable. Average per-packet latency depends mostly on sanitization rule complexity: how much processing and memory read/write operations it performs. Having multiple rules loaded in the developed packet sanitizer scales the processing latencies linearly with the number and complexity of enabled rules if those set of loaded rules actually are triggered by the incoming and matched packets. The no-op rules have negligible impact on the overall processing latencies and throughput.

Even though the packet processing latency evaluations were performed on a conventional hardware system (AMD Ryzen 5 PRO 5650U, Linux 6.8, XDP generic mode with JIT enabled), the implementation sustained high packet processing rates - hundreds of thousdans packets processed per second. The eBPF/XDP sanitizer outperformed an equivalent netfilter's NFQUEUE user-space implementation by approximately 5 times for comparable packet sanitization workloads. These results indicate that moving packet processing and sanitization logic in kernel at the earliest possible packet processing point can substantially reduce processing overhead and improve throughput while providing consistent per-packet latency behaviour.

There are important limitations that qualify these results. Tests used synthetic traffic with a same physical host for packet generator and packet sanitizer and exercised the eBPF program in XDP hook in generic mode. Results may differ on other hardware, with more performant native/driver XDP support, or with real-world malicious traffic mixes. The current dataplane targets IPv4 and stateless TCP/UDP header sanitization and stores rules in fixed-size arrays (currently up to 64 entries per protocol) to satisfy eBPF verifier constraints: these trade-offs affect potential scalability and feature scope.

In summary, this project demonstrates that a carefully constrained, eBPF-based packet sanitizer is a practical and performant approach for inline packet sanitization even on conventional hardware. By prioritizing verifier friendliness and predictable dataplane structure, the implementation achieves low and stable per-packet cost while remaining extendable via bpf-tail calls and its control plane. With additional validation on diverse hardware and expanded rule expressiveness, the approach is well-positioned for potential production use.

# Future work

While currently developed eBPF/XDP packet sanitizer system is sufficinently performant for potential utilization in production systems, some future work for further extension and improvement of the described packet sanitization engine, rules and methodology could be excercised. The following list lays out the potential topics for future work:

- Research and development of deep packet inspection (DPI) capabilities in the sanitizer system to analyze packet payloads beyond header information. Performing complex DPI will require efficient offloading into the user-space and would not be sustainable for all the incoming traffic. Implementation of efficient packet sampling techniques for DPI performed in user-space in order to comply with tight requirements with eBPF program size and verifier constraints which prevent of arbitrary memory reads and complex logic would be required for peforming real complex DPI.

- Support for lightweight connection tracking capabilities for connection-oriented protocols such as TCP. Implementing connection state tracking within eBPF maps to maintain session information across multiple packets or utilizing traffic control hooks to determine connection establishment without additionally managing connection state information at the XDP level. The system would benefit from enhanced TCP sequence number validation, connection establishment monitoring, and session hijacking detection capabilities.

- Integration of lightweight machine learning models into the eBPF-based packet sanitization engine or potentially in user-space components to enable adaptive and intelligent packet filtering based on learned traffic patterns of live running system.

- Direct higher level OSI layers protocol suppport, such as application level HTTP/2, HTTP/3, DNS and similar. This would allow for even more granular sanitizations and inspections taylored to specific application protocols and workloads.

# References


[1] 18.1. lua usage in suricata - suricata 8.0.0-rc1-dev documentation. [Online; accessed 2025-05-27].

[2] Bpf maps - the linux kernel documentation. [Online; accessed 2025-06-06].

[3] Bpf ring buffer - the linux kernel documentation. [Online; accessed 2025-06-06].

[4] cilium/ebpf: ebpf-go is a pure-go library to read, modify and load ebpf programs and attach them to various hooks in the linux kernel. [Online; accessed 2025-05-31].

[5] Home - suricata. [Online; accessed 2025-05-27].

[6] libnetfilter_queue: Main page. [Online; accessed 2025-05-28].

[7] Netfilter conntrack memory usage - john leach. [Online; accessed 2025-05-25].

[8] netfilter/iptables project homepage - the netfilter.org project. [Online; accessed 2025-05-24].

[9] Programmer's guide - data plane development kit 25.03.0 documentation. [Online; accessed 2025-05-25].

[10] Snort - network intrusion detection & prevention system. [Online; accessed 2025-05-26].

[11] Tail calls - ebpf docs, 2025.

[12] David Day and Benjamin Burns. A performance analysis of snort and suricata network intrusion detection and prevention engines. 02 2011.

[13] Luca Deri, Netikos A, Via Km, and La Loc. Improving passive packet capture: Beyond device polling. 09 2004.

[14] Talaya Farasat, JongWon Kim, and Joachim Posegga. Smartx intelligent sec: A security framework based on machine learning and ebpf/xdp, 2024.

[15] Matt Fleming. A thorough introduction to ebpf [lwn.net], December 2017.

[16] florianl. florianl/go-nfqueue: c-binding free api for golang to communicate with the queue subsystem of netfilter. [Online; accessed 2025-06-09].

[17] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 29–38, 2015.

[18] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. The ebpf runtime in the linux kernel, 2024.

[19] Peter N. M. Hansteen. *The Book of PF: A No-Nonsense Guide to the OpenBSD Firewall*. No Starch Press, 3 edition, 2014.

[20] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. page 54–66, 2018.

[21] Shuwen Liu, Yu Liu, and Craig A. Shue. Inspecting traffic in residential networks with opportunistically outsourced middleboxes. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2023.

[22] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[23] Raik Niemann, Udo Pfingst, and Richard Göbel. Performance evaluation of netfilter: A study on the performance loss when using netfilter as a firewall, 2015.

[24] OpenBSD Project. Pf user's guide, 2025. [Online; accessed 2025-11-02].

[25] OpenBSD Project. Pf user's guide - stateful tracking, 2025. [Online; accessed 2025-11-02].

[26] OpenBSD Project. Pf user's guide - traffic normalization, 2025. [Online; accessed 2025-11-02].

[27] OpenBSD Project. pf.conf - packet filter configuration file, 2025. [Online; accessed 2025-11-02].

[28] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9, USA, 2012. USENIX Association.

[29] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[30] K. Salah and A. Kahtani. Performance evaluation comparison of snort nids under linux and windows server. *Journal of Network and Computer Applications*, 33(1):6–15, 2010.

[31] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), February 2020.

[32] Shie-Yuan Wang and Jen-Chieh Chang. Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, 198:103283, 2022.