sheriff-buzz

Port Scan Detection and Response with eBPF

Salma Berriche

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

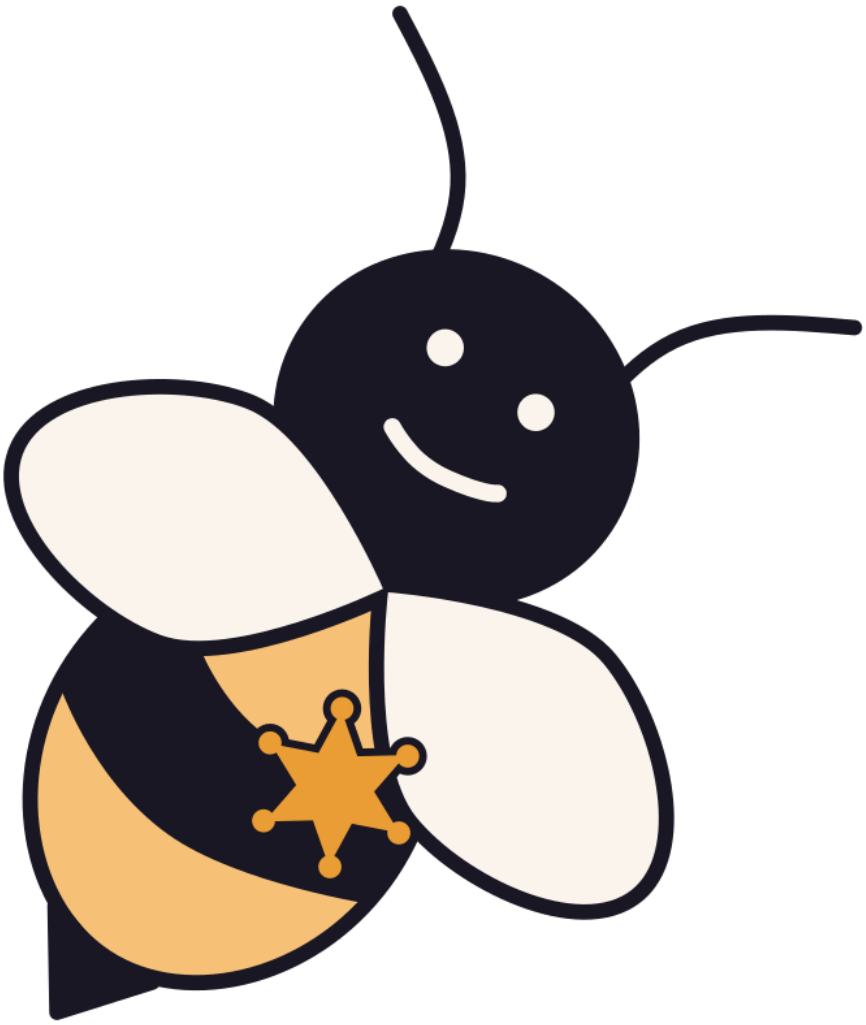
eBPF (extended Berkeley Packet Filter) has emerged in recent years as the de facto mechanism for extending and modifying Linux (and increasingly other operating systems') kernel behaviour without the need for modifying source code or writing kernel modules. Running in the kernel makes it well-suited for high performance packet processing.

Port scans are an undesirable but unavoidable type of network traffic; a new device connected to the Internet is typically scanned within minutes. They are fundamental tools for attackers probing for weaknesses, making them one type of activity that Intrusion Detection Systems (IDS) are designed to identify.

In this project, I implement `sheriff-buzz`: a port scan detection and response system using eBPF.

Acknowledgements

I would like to thank my supervisor Eike Ritter for his invaluable guidance over the course of the project, my parents for their unwavering support, and my sister Leyla for designing the adorable `sheriff-buzz` logo.



`sheriff-buzz` logo

Abbreviations

eBPF	Extended Berkeley Packet Filter
IDS	Intrusion Detection System
BTF	BPF Type Format
CO-RE	Compile Once, Run Everywhere
XDP	eXpress Data Path
TC	Traffic Control
LRU	Least Recently Used

Contents

List of Figures

List of Tables

Chapter 1  Introduction

eBPF is a technology that allows running small sandboxed programs in kernel space [1]. It was originally developed in Linux [2] as an extension to the Berkeley Packet Filter [3] but has since been ported to Microsoft Windows [4] and is under active development for MacOS[1] and FreeBSD [7].

eBPF adds event-driven programmability to the kernel; eBPF programs can be attached to kernel events (network packet arrival, system calls, function entries and exits) to collect telemetry, filter network traffic, enforce security policies etc., all without the need to modify kernel source code or load external kernel modules.

1.1 Motivation

The motivation for writing a port scan detection system using eBPF is to use its unique capabilities for efficient and safe system monitoring within the Linux kernel.

eBPF's main advantages over modifying kernel source or developing kernel modules are:

- **Safety and stability**: eBPF programs are checked by the in-kernel eBPF verifier prior to loading to ensure they don't crash the kernel, have bounded execution time (no infinite loops), and can only access allowed kernel functions and memory regions unlike kernel source code and modules that have full privileges so can crash the entire system, cause data corruption, and introduce security vulnerabilities.
- **Reduced maintenance overhead**: eBPF programs can be loaded and unloaded dynamically without having to reboot the system. CO-RE (Compile Once - Run Everywhere) makes eBPF programs less dependent on specific kernel versions, eliminating the need for recompiling when the kernel is updated.
- **Fast development and deployment**: The development cycle is typically faster than that of core kernel code and kernel modules given that eBPF programs are small and focused, making them easier to alter. eBPF programs can be loaded without kernel recompilation or machine reboots.

eBPF's main advantages over user space applications:

- eBPF runs directly in the kernel, thus eliminating the overhead of context switches.
- eBPF programs are JIT (Just-In-Time) compiled thus achieving near-native execution speed.
- eBPF programs can directly access kernel data structures, functions, and trigger on low-level kernel events (tracepoints, kprobes, network packet paths), which allows for highly granular low-latency visibility.
- eBPF allows for efficient data processing in the kernel, thus minimising data transfer between kernel and user space, which results in increased security and lower overall CPU and memory usage.

1.2 Project Aim

Malicious actors—both from inside and outside the network—attempt to breach network defences through various methods such as port scanning, vulnerability exploitation, denial of service attacks, malware delivery, etc. An IDS program monitors network traffic looking for suspicious patterns that may indicate an attempted intrusion and act as per system-defined policy.

The aim of this project is to leverage eBPF capabilities to develop `sheriff-buzz`, a port scan detection and response program.

[1] MacOS does not have a kernel-level eBPF subsystem, but there are several methods of running eBPF programs such as virtual machines, Docker, and user space libraries [5, 6].

Chapter 2 Background and Research

In this chapter I discuss the eBPF architecture, more specifically the virtual machine and the Linux kernel eBPF verifier, explain how CO-RE works, and introduce XDP. I briefly describe port scanning as a method of discovering potential entry points into a system. In closing I mention a few existing systems and reference related network security research.

2.1 eBPF in the Linux Kernel

eBPF programs (bytecode) are loaded into the kernel using the `bpf()` system call and go through the following stages (see Fig. 2.1):

- **compilation**: the C code is compiled into eBPF virtual machine bytecode using LLVM [8]
- **verification**: the eBPF verifier does a static analysis of the bytecode
- **JIT compilation**: the bytecode is JIT (Just-In-Time) compiled into a native machine code program
- **attaching**: the program is attached to the specified kernel hook (e.g. tracepoint, kprobe)

2.1.1 Verifier

The eBPF verifier performs static analysis to ensure that loaded programs are safe to run [9].

The verifier sanity checks the program in two stages:

- **DAG check:** depth-first search to parse the bytecode into a *directed acyclic graph*. This is meant to detect infinite loops and unreachable instructions.
- **FSM check:** simulate execution of all possible code paths using a *finite state machine* while checking the contents of the registers and the stack. The verifier prunes branches with states (registers and stack contents) it has already encountered and deemed safe.

The limited instruction set of the eBPF virtual machine and the relatively small stack size (512 bytes) enforce a complexity limit that ensures the verification process itself completes in reasonable time.

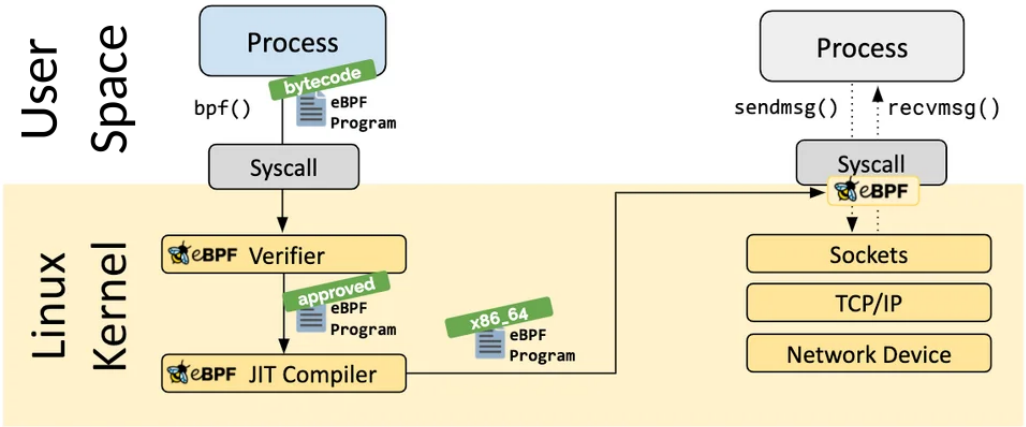


Fig. 2.1: eBPF program loading workflow [1]

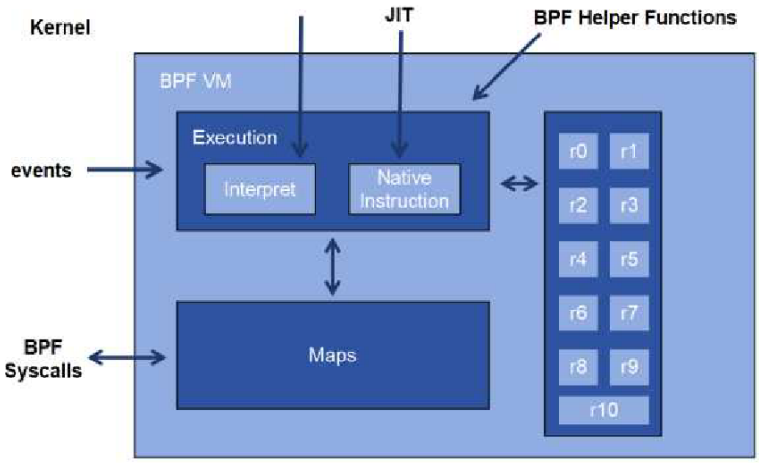


Fig. 2.2: The eBPF virtual machine [11]

Register(s)	Purpose
`r0`	Holds the function return value
`r1-r5`	Used to pass function parameters
`r6-r9`	Preserved across function calls
`r10`	Read-only frame pointer
`r11`	Program counter

Table 2.1: eBPF virtual machine registers

2.1.2 JIT Compilation

Once the eBPF virtual machine bytecode passes the verifier, it is JIT (Just-In-Time) compiled into native machine code by the in-kernel compiler. This was not always the case; the bytecode used to be interpreted which is slower than the current implementation [1, 10].

2.1.3 Attaching

eBPF programs are event-driven: they attach to kernel events such as system calls, tracepoints, and function entries/exits. For example, an eBPF program that is attached to the `open()` system call will see its JIT-compiled code run every time a process calls `open()`.

2.1.4 eBPF Virtual Machine

The purpose of the eBPF virtual machine is to provide a safe, verified, and efficient execution environment inside Linux kernel space. It is effectively the sandbox for eBPF programs.

The eBPF virtual machine instruction set is small and simple by design (see Fig. 2.2 and Table 2.1), making it easily analysable by the verifier. The instruction set is also designed to be efficiently translated into native machine code using a JIT compiler, enabling eBPF programs to run with very little overhead.

The eBPF virtual machine provides a stable instruction set that offers a layer of abstraction between eBPF programs and the underlying hardware architecture.

2.1.5 eBPF CO-RE

CO-RE (Compile Once - Run Everywhere) is the mechanism used by eBPF to achieve portability [12]. Kernel data structures and functions can change in between kernel versions which, in theory, would require eBPF programs to be recompiled.

CO-RE requires the cooperation of the following components:

- **BTF (BPF Type Format)**:
  - metadata describing kernel data types (structs, unions, functions etc.)

  - the target kernel needs to have been compiled with BTF enabled (`CONFIG_DEBUG_INFO_BTF=y`)
  - kernel BTF information is available at `/sys/kernel/btf/vmlinux`
- **Compiler**:
  - the C code needs to have been compiled with `-g` to include BTF information for the eBPF program
  - LLVM generates relocation records within the compiled eBPF object file instead of hardcoding memory offsets
- **Runtime adjustments**:
  - when the eBPF bytecode runs on the target machine, `libbpf` [13] reads the BTF and relocation records from the eBPF object file
  - `libbpf` then reads the BTF information from the running kernel and compares it with that of the eBPF program
  - `libbpf` calculates the adjusted offsets and sizes of the structure fields based on the kernel's BTF information
  - `libbpf` patches the eBPF bytecode in memory by updating the instructions with the correct offsets prior to handing it to the verifier

2.1.6 eBPF Maps

Programs have a stack size limit of 512 bytes, so eBPF maps are used for persistent storage and communication between kernel and user space as well as eBPF programs themselves. There are several types of map, notably hash tables, arrays, and ring buffers.

2.1.7 XDP

eBPF programs have types that affect:

- the points that they are allowed to attach to
- what data they receive as an argument when triggered (*context data*)
- which kernel functions they are permitted to call
- the interpretation of their return value

`BPF_PROG_TYPE_XDP`—often referred to as XDP (eXpress Data Path)—is one such type. An XDP program attaches to a network interface and intercepts packets before they reach the OS network stack [14]. The program processes incoming packets, and decides the action to be taken for them. For example, `XDP_PASS` allows the packet to flow through the network stack and `XDP_DROP` drops the packet.

2.2 Port Scans

A port scan is a technique used to identify open ports on a network host [15]. When conducted with malicious intent, they scan targets in order to discover potential entry points, identify potentially vulnerable services, and gather information about the target network topology. Port scans are often used as the reconnaissance stage of a more focused attack.

It is potentially illegal to perform a port scan without explicit permission from the target network; the act of performing one can be seen as an attack within itself in this situation [16].

2.2.1 Types of Port Scan

- **TCP SYN Scan**: Also known as a half-open scan or stealth scan. Send a SYN packet as if opening a connection, using the response to determine if the port is open. The default `nmap` scan.
- **TCP Connect Scan**: Establish a connection (perform the TCP handshake) with each port in order to obtain status information. Used when a SYN scan is not possible.
- **Unusual flag combinations**: Typically used to make the scan go undetected by firewalls and/or logging systems.

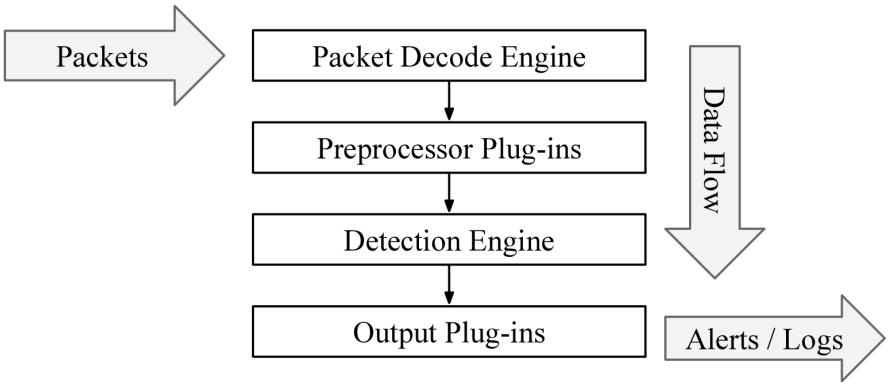


Fig. 2.3: Snort architecture [22]

  - **NULL scan**: do not set any flags
  - **FIN scan**: set `FIN` flag only
  - **Xmas scan**: set `FIN`, `PSH`, and `URG` flags, "lighting the packet up like a Christmas tree" [16]
- **UDP Scan**: Send a UDP packet to each port, using the response (or lack thereof) in order to determine whether it is open, potentially filtered, or closed.

`nmap` [17] is a widely used network exploration tool capable of port scanning. It allows for customisation including setting the range and number of ports, delay between packets, and TCP header flags etc.

2.3 Existing Systems

There are several modern intrusion detection systems capable of detecting and responding to port scans. I chose to mention two open-source tools: the classic user space Snort and the more modern eBPF-based Suricata.

2.3.1 Snort

Snort is an "open source intrusion prevention system capable of real-time traffic analysis and packet logging" [18]. It provides real-time network analysis including port scan detection; its Network Intrusion Detection System Mode uses a configurable rule system to perform actions on matching packets [19]. Fig. 2.3 shows Snort's packet processing algorithm: packets are decoded, run through preprocessors, and then the detection engine. For example, the "reputation" preprocessor can be used to specify IP blacklists and whitelists.

Snort's configuration (written in the Lua scripting language [20]) allows for setting of port scan thresholds [21]. Setting thresholds helps avoid false positives and negatives as it allows a user to adjust them based on expected legitimate traffic patterns. Snort does not use eBPF for packet processing, instead it relies on the Linux kernel networking stack and mechanisms such as Netfilter Queues (NFQ) to intercept packets.

2.3.2 Suricata

Suricata is a "high performance Network IDS [Intrusion Detection System], IPS [Intrusion Prevention System] and Network Security Monitoring engine" [23]. It uses eBPF for packet filtering, load balancing, and loading custom XDP programs. The XDP bypass feature [24] allows for dropping of packets via XDP (before they reach the network stack), giving a performance advantage over non-eBPF tools such as Snort.

Similar to Snort, Suricata supports rules as well as more advanced scripting with Lua. However, its IP reputation system allows for finer-grained control, allowing for IP categorisation and scoring.

2.4 Related Work

eBPF is an active research field; the technology is constantly being developed along with new applications, including non-networking uses. This section primarily focuses on XDP program performance analysis and the use of eBPF in network intrusion detection.

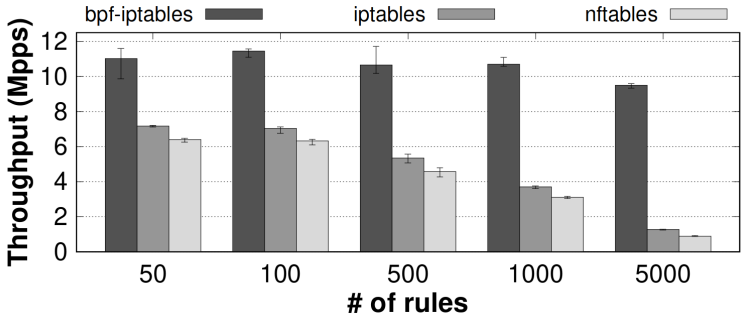


Fig. 2.4: Throughput with varying numbers of `iptables`, `nftables`, and `bpf-iptables` firewall rules [25]

XDP has been shown to be more performant than in-kernel firewalls `iptables` and `nftables` thanks to its earlier packet access. [25] proposes `bpf-iptables`—an eBPF-based port of `iptables`—and compares its performance to `iptables` and `nftables`. It uses XDP for incoming packets and the later TC (Traffic Control [26]) hook for outgoing packets (XDP does not work on outgoing packets). Fig. 2.4 shows the throughput in million packets per second with respect to number of rules of the three firewall implementations; `bpf-iptables` greatly outperforms the field, and its performance is barely negatively affected as the number of rules increases.

[27] presents and compares two methods of detecting flag-based `nmap` scans: `netfilter` and eBPF/XDP, finding that the latter implementation achieved a higher throughput (up to 5% increase). Both implementations perform a series of checks on the TCP headers and drop packets if they contain suspect TCP flag combinations (e.g. Xmas, FIN, NULL), potentially yielding false positives due to the lack of thresholding. The proposed system is limited to detecting port scans with these predefined TCP flag combinations.

The authors of [22] propose an eBPF-based intrusion detection system compatible with modified Snort rulesets. It follows Snort's detection engine architecture, using an eBPF fast pattern matching engine and a rule tree matching engine running in user space. The system has been shown to outperform Snort by a factor of 3. The authors blamed verifier restrictions for limiting the eBPF component to a simple set of rules. However, I believe that an implementation that runs the detection engine in user space and only packet processing as an eBPF program in kernel space would eliminate the problem identified by the authors.

eBPF-LAIN is an intrusion detection system using machine learning to detect port scans [28]. It uses a random forest model trained with features such as number of bytes sent, TCP flags, and connection duration. It attempts to avoid false positives by taking into account number of packets and the delay between them, both of which are configurable. eBPF-LAIN performs port scan detection in kernel space and stores malicious traffic information in user space.

Chapter 3  System Requirements

This chapter outlines the functional and non-functional requirements for `sheriff-buzz` using the MoSCoW model [29], which defines four priority levels, from most to least critical:

- **Must Have (M):** Critical; must be included in the final product
- **Should Have (S):** High priority; should be included if possible
- **Could Have (C)**: Desirable/nice-to-have; could include if there is time
- **Won't Have (W):** Out-of-scope; will not be included

Assigning priorities to requirements in this way allows development to be focused on vital features. Setting W requirements defines the scope of the project and prevents feature creep.

3.1 Functional Requirements

The functional requirements define the desired features of `sheriff-buzz`. It is critical that port scan detection and minimal response functionality are available along with informative logging. Detection and response features must also be configurable to fit a wide range of user requirements. Other high-priority features include packet redirection and the ability to blacklist and whitelist sources. Whitelisting destination ports, a dry run mode, and on-demand statistics reporting are also desirable.

Functional Must Have (FM)

ID	Name	Description
FM1	Port scan detection	detect port scans based on suspect TCP flag combinations and/or abnormally large numbers of ports
FM2	IP blocking	drop traffic from flagged IP addresses
FM3	Logging	log error messages, optional debug statements, and alerts
FM4	Alert database storage	store alert information in a database table
FM5	Blocked IP database storage	store blocked IP addresses in a database table
FM6	User configuration	support user-supplied configuration

Functional Should Have (FS)

ID	Name	Description
FS1	IP redirection	redirect traffic from flagged IPs to a user-specified target (e.g. a honeypot)
FS2	IP blacklist/whitelist	support specifying blacklisted/whitelisted IPs from a config file
FS3	Subnet blacklist/whitelist	support specifying blacklisted/whitelisted subnets from a config file

Functional Could Have (FC)

ID	Name	Description
FC1	Port whitelist	support specifying whitelisted destination ports from a config file
FC2	Dry run mode	support a mode that reports alerts without taking further action on traffic
FC3	Statistics reporting	output program statistics (e.g. packets processed per second) on request
FC4	Config reload	automatically reload configuration upon modification

Functional Won't Have (FW)

ID	Name	Description
FW1	Port blacklist	do not support specifying a blacklisted destination port from a config file
FW2	IP blacklist aggregation	do not correlate activity from multiple IPs back to a common subnet
FW3	IPv6 support	no support for IPv6 network traffic

3.2 Non-functional Requirements

An effective IDS must be resource-efficient and scalable to accommodate network growth and varying traffic loads. Documentation and clear error messages are also a must. `sheriff-buzz` should be resilient to unexpected database component failure. Ideally the program should support reconnecting to the database instead of requiring a complete restart. CO-RE ensures the program is portable, therefore it must be implemented. `sheriff-buzz` will not have a graphical user interface.

Non-functional Must Have (NM)

ID	Name	Description
NM1	Scalability	process concurrent traffic from large numbers of IPs
NM2	Resource efficiency	lightweight daemon; use as little CPU and memory resources as possible
NM3	User documentation	document command-line arguments and configuration file options
NM4	Database error resilience	database-related errors are not fatal to the main program
NM5	CO-RE	`sheriff-buzz` can be deployed on compatible kernels without the need to recompile

Non-functional Should Have (NS)

ID	Name	Description
NS1	Developer documentation	document parameters and return values of functions for code maintainability
NS2	Clear error messages	informative error messages

Non-functional Could Have (NC)

ID	Name	Description
NC1	Database reconnection	attempt to reconnect to the database if connection is lost during execution

Non-functional Won't Have (NW)

ID	Name	Description
NW1	GUI	no graphical user interface; the system is better suited to run on the command-line, ideally as a background daemon or, better yet, a system service

Chapter 4 Design

In this chapter I detail the design of `sheriff-buzz` that satisfies the requirements outlined in the previous chapter. I start by describing the overall system architecture before delving into how each component works.

4.1 System Architecture

`sheriff-buzz` has a kernel space eBPF component and a user space component.

The eBPF component processes network packets and coordinates with the user space component to decide the proper course of action. The user space component analyses packets and flags port scans, sharing its findings with the eBPF component. It records alert incident details in a database. It is also responsible for reloading program configuration whenever the user modifies the associated file on disk.

Fig. 4.1 summarises the system architecture of `sheriff-buzz`.

4.2 Packet Processing with eBPF

The eBPF component intercepts packets on the monitored network interface and determines the subsequent action based on an evaluation of the source IP address, source subnet, and destination port.

The possible results of the evaluation are:

Name	Action	Description
blacklist	drop/redirect	Drop the packet or redirect it to a user-specified IP (depending on configuration).
whitelist	pass	Pass the packet on to the network stack. This allows users to whitelist IPs, subnets, and destination ports so `sheriff-buzz` does not analyse traffic originating from/meant for them.
unknown	pass	Pass the packet on to the network stack *and* send headers to user space for analysis.

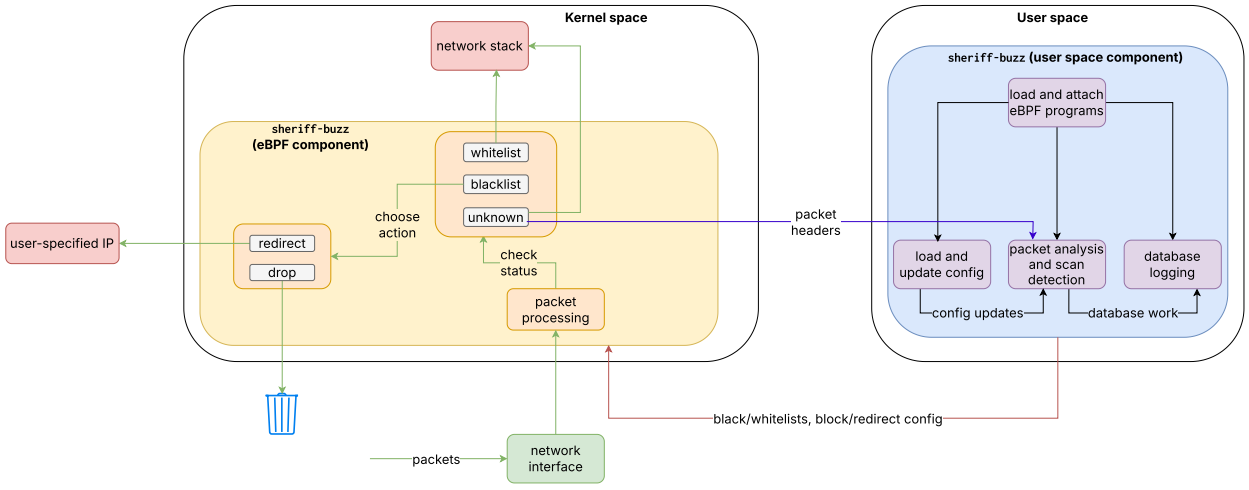


Fig. 4.1: `sheriff-buzz` architecture

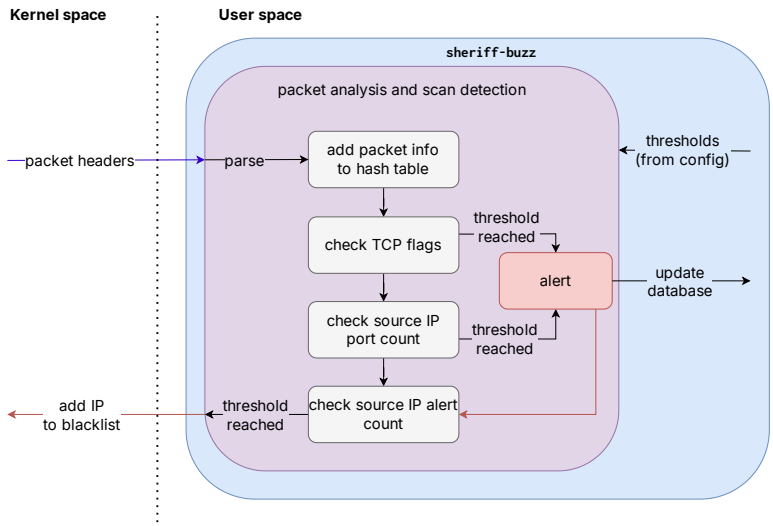


Fig. 4.2: Packet analysis component design

4.3 Packet Analysis and Scan Detection in User Space

I opted for all the analysis to take place in user space which is better suited for the task as it avoids the ad-hoc limitations linked to the eBPF design (e.g. stack size limit, bounded loops etc.). This approach also allows for future complex filtering rules without slowing down in-kernel packet processing. This is a different design than those in [22, 27, 28] which carry out some or all of the packet processing and intrusion detection work in eBPF at the cost of only being able to use simple rules. Fig. 4.2 outlines the design of the user space packet analysis component.

4.3.1 Packet Hash Table

Similar to Snort [18], `sheriff-buzz` stores information about packets in a hash table to allow for efficient updates and lookups.

4.3.2 Scan Detection

There are two types of scan detection criteria:

- **Flag-Based**: TCP packet flags are FIN, NULL, or Xmas (see Section 2.2.1).
- **Port-Based**: source IP sends packets to a number of ports greater than a configurable threshold

4.3.3 Thresholds

`sheriff-buzz` uses packet, port, and alert thresholds to avoid false positives (Table 4.1). For instance, the user can configure the port threshold to be comfortably larger than the number of ports they expect legitimate traffic on.

4.4 Configuration

Users can use a config file to fine-tune alert detection and response as they see fit. Table 4.2 outlines the supported configuration options.

`sheriff-buzz` reloads the configuration whenever the config file is modified while running without the need for a restart.

4.5 Database Logging

`sheriff-buzz` records alert and blocked IP information in a database across the following tables (see Fig. 4.3 for more details):

Name	Description	Decision
Packet threshold	*(Flag-based scans only)* A packet with a suspect combination of TCP flags triggers the comparison of the total number of packets (sent by the source so far) to the packet threshold.	Total number of packets $\geq$ packet threshold $\implies$ port scan alert
Port threshold	*(Port-based scans only)* Number to which the total number of ports an IP sends packets to is compared.	Total number of ports $\geq$ port threshold $\implies$ port scan alert
Alert threshold	Number of alerts triggered for a given source IP before it is blacklisted.	Total number of alerts $\geq$ alert threshold $\implies$ blacklist IP

Table 4.1: Types of scan detection threshold

Name	Description
Action	Action to take for packets from blacklisted sources (either block or redirect)
Thresholds	Scan detection thresholds- see Table 4.1
Redirection IP	IP address to redirect traffic to (if applicable)
Blacklists	Sources to block/redirect (depending on action)
Whitelists	Sources to pass packets on to the rest of the network stack (with no analysis) from

Table 4.2: Configuration options

- **`alert_type`** contains the supported types of alert `sheriff-buzz` uses; this can be extended in future work
- **`scan_alerts`** contains alert information with the `alert_type` ID as a foreign key
- **`blocked_ips`** contains IPs blocked by `sheriff-buzz`

The user can query `scan_alerts` by alert type, source IP etc. and `blocked_ips` by source IP and time. The user can use the insights gained from the database to fine-tune `sheriff-buzz` configuration.

4.6 Inter-Component Communication

The design uses the following components:

- **Kernel space**
  - **Packet Processing**: the eBPF program (Section 4.2)
- **User space**

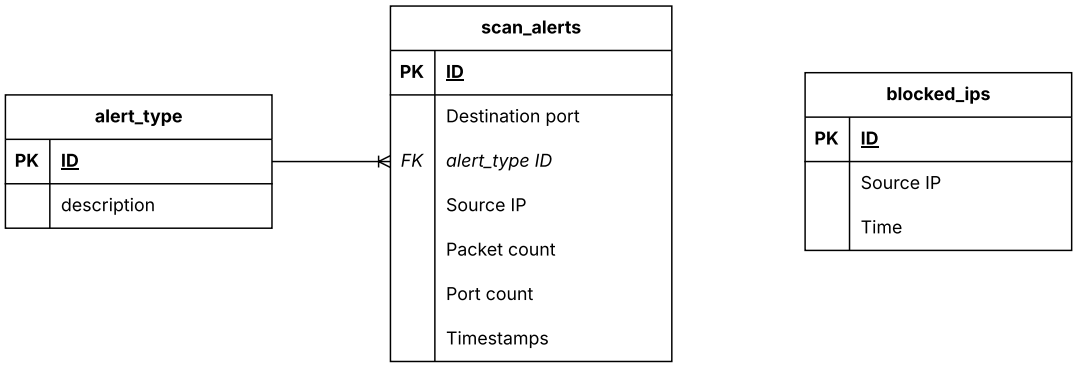


Fig. 4.3: Entity relationship diagram

- **Analysis and Scan Detection**: responsible for detecting port scans (Section 4.3)
- **Configuration**: responsible for automatically reloading user config upon modification (Section 4.4)
- **Database Logging**: responsible for all database-related work (Section 4.5)

The following table describe the existing communication channels between the components:

From	To	What	Why
Packet Processing	Analysis and Scan Detection	packet headers	decide if a port scan has taken place
Configuration	Packet Processing	blacklists, whitelists, and blacklist action from config file	update IP, subnet, and port information and corresponding actions
Analysis and Scan Detection	Packet Processing	IPs blacklisted as a result of traffic analysis	update IP information
Analysis and Scan Detection	Database Logging	alert and blocked IP data	update database

Chapter 5  Implementation

In this chapter I describe the implementation details of `sheriff-buzz`.

5.1 System Architecture

I chose to implement `sheriff-buzz` in C for the following reasons:

- **Better eBPF tooling**:
  - the primary toolchain for compiling eBPF programs is based on LLVM, which has mature and robust C support
  - `libbpf` [13]—one of the core eBPF libraries—is a C library, which makes integration straightforward when the eBPF code is written in C
- **Performance**:
  - C allows fine-grained control over memory and low execution overhead. This is what results in highly efficient bytecode
  - eBPF tasks involve manipulating kernel data structures, network packet data etc. C is well suited for these low-level operations
- **Proximity to the Linux Kernel**:
  - given the Linux kernel itself is predominantly written in C, writing the user space program in C makes data types, helper functions etc. align closely with their kernel counterpart
  - writing the eBPF and user space components in C makes sharing header files between the two seamless

5.1.1 eBPF Component

I implemented five eBPF programs:

- **`process_packet`** (of type `BPF_PROG_TYPE_XDP`): attaches to a network interface; runs for each incoming packet on this interface. The return code determines the action to be taken with the packet (Section 5.3).
- **`read_{ip,port,subnet,config}_rb`** (of type `BPF_PROG_TYPE_KPROBE`): referred to as *uretprobes* (Section 5.4).
  - *u* stands for user space; the probe targets a function running in user space. In this case, a function inside `sheriff-buzz` itself.
  - *ret* stands for return; the probe triggers when the specified user space function is about to return to its caller.
  - *probe* is the mechanism of attaching an eBPF program to this type of event i.e. a function

I will refer to `BPF_PROG_TYPE_XDP` as **XDP**, and `BPF_PROG_TYPE_KPROBE` as **uretprobe** for the remainder of the document.

Fig. 5.1 shows how the eBPF programs interact to process network packets. Fig. 5.2 shows the listing produced by `bpftool prog show` when `sheriff-buzz` is running.

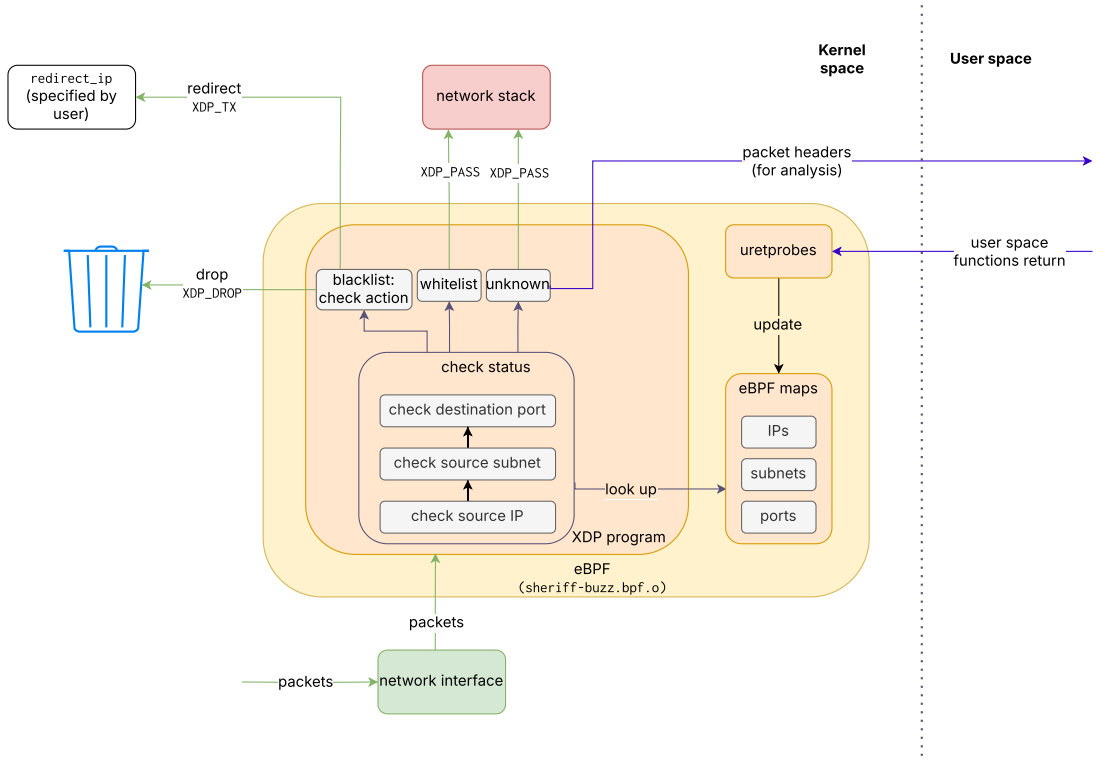


Fig. 5.1: Structure of the eBPF component of `sheriff-buzz`

```
# bpftool prog show
7710: xdp  name process_packet  tag 659ccbadff5a1555  gpl run_time_ns 883152085 run_cnt 259456
        loaded_at 2025-04-10T22:55:44+0100  uid 0
        xlated 2560B  jited 1542B  memlock 4096B  map_ids 5248,5240,5241,4495,5242,5243
        btf_id 9138
        pids sheriff-buzz(51956), bpftool(102683)
7705: kprobe  name read_ip_rb  tag e7558d8a68c026cf  gpl run_time_ns 431794 run_cnt 54
        loaded_at 2025-04-10T22:55:44+0100  uid 0
        xlated 176B  jited 120B  memlock 4096B  map_ids 5244,5240
        btf_id 9138
        pids sheriff-buzz(51956)
7706: kprobe  name read_subnet_rb  tag 665e1564a0e1982a  gpl run_time_ns 1465 run_cnt 1
        loaded_at 2025-04-10T22:55:44+0100  uid 0
        xlated 232B  jited 152B  memlock 4096B  map_ids 5245,5241
        btf_id 9138
        pids sheriff-buzz(51956)
7707: kprobe  name read_port_rb  tag f47ccab57cbbbdb9  gpl
        loaded_at 2025-04-10T22:55:44+0100  uid 0
        xlated 176B  jited 120B  memlock 4096B  map_ids 5246,5242
        btf_id 9138
        pids sheriff-buzz(51956)
7708: kprobe  name read_config_rb  tag 1b26afd1cc03c8eb  gpl run_time_ns 3176 run_cnt 1
        loaded_at 2025-04-10T22:55:44+0100  uid 0
        xlated 184B  jited 130B  memlock 4096B  map_ids 5247,5248
        btf_id 9138
        pids sheriff-buzz(51956)
```

Fig. 5.2: eBPF programs listed with `bpftool`

5.1.2 User Space Component

The user space component consists of three `pthreads`:

- **Main Thread**: loads and attaches eBPF programs then installs an **event handler** that runs a callback function (`handle_event()`) whenever it receives packet headers from `process_packet`. (Section 5.5)
- **Database Worker (`db_worker`)**: writes alert and blocked IP information to the database (Section 5.8).
- **Config Worker (`inotify_worker`)**: watches for changes to the config file and syncs it to the in-memory config (Section 5.7).

The main thread and database worker were initially a single thread. I later split them into two threads since they are intended to perform independent sets of tasks and therefore lend themselves naturally to a multithreaded model. This decision was validated in the significant performance improvement I observed.

5.2 Loading eBPF Programs

The main thread of the user space component loads and attaches the eBPF programs into the kernel. This could have been achieved using either BCC [30] or `libbpf` [13].

BCC compiles eBPF source code into bytecode at runtime thus making the program not CO-RE as it requires kernel headers and BCC to be present on the target machine. On the other hand, `libbpf` is CO-RE aware (see Section 2.1.5) by design [12]. Moreover, the application startup is much faster since the application code is pre-compiled in the case of `libbpf` compared to BCC that needs to perform compilation at runtime. For these reasons I opted to use `libbpf` instead of BCC.

Loading the eBPF programs into the kernel initialises the eBPF maps used to store blacklists and whitelists of IP addresses, subnets, and ports. Additional eBPF maps are used to store configuration and allow event-based communication between kernel and user space components. These are discussed in more detail in Sections 5.3.2, 5.4, and 5.10.

5.3 XDP Packet Processing

The `process_packet` XDP program examines incoming packets on the monitored network interface. It accesses the raw packet data as a `struct xdp_md` object.

`process_packet` uses the source IP, source subnet, and destination port to determine the appropriate action for a packet with an order of precedence:

- source IP address
- source subnet
- destination port

If the action is unknown, it sends the packet headers to the user space component for analysis.

See the flowchart in Fig. 5.3 for an illustration.

5.3.1 Extracting Packet Headers

The XDP program performs length checks on the `struct xdp_md` context to guard against out-of-bounds memory accesses that may be caused by malformed packets. For example, Fig. 5.4 shows how the code ensures that that the packet data is large enough to contain Ethernet, IP, and TCP headers prior to extracting the TCP headers. The eBPF verifier rejects a program lacking such checks; see Fig. 5.5 for a typical error message returned by the verifier in such scenarios.

5.3.2 eBPF Map Storage

The eBPF component stores blacklisted and whitelisted IPs and destination ports in LRU hash maps so that the least recently used entries are evicted when the map is full [31]. I chose to use LRU hash maps to solve the inevitable problem of the maps filling up when `sheriff-buzz` runs for an extended period of time (months or even years) under heavy network traffic.

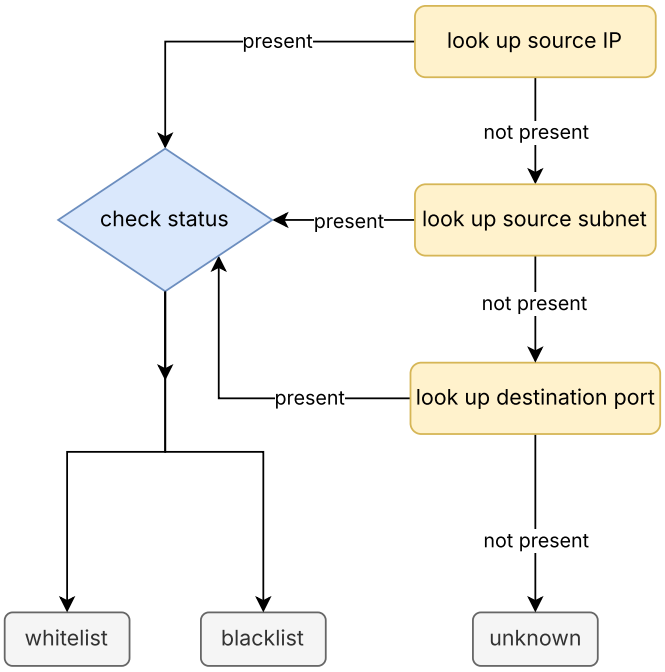


Fig. 5.3: Packet action flowchart

```
struct xdp_md *ctx;
...
if ((char *) ctx->data + sizeof(struct ethhdr) + sizeof(struct iphdr)
                + sizeof(struct tcphdr) > (char *) ctx->data_end)  {
        goto fail;
}
...
```

Fig. 5.4: XDP data length check before extracting TCP headers

```
libbpf: prog 'process_packet': BPF program load failed: Permission denied
libbpf: prog 'process_packet': -- BEGIN PROG LOAD LOG --
...
; if (lookup_protocol(ctx) == TCP_PNUM) { @ sheriff-buzz.bpf.c:202
183: (55) if r2 != 0x6 goto pc-97     ; R2_w=6
; if (dst_port_state(tcp_headers->dest) == WHITELIST) { @ sheriff-buzz.bpf.c:640
184: (69) r1 = *(u16 *)(r1 +36)
invalid access to packet, off=36 size=2, R1(id=0,off=36,r=34)
R1 offset is outside of the packet
processed 236 insns (limit 1000000) max_states_per_insn 2 total_states 22 peak_states 22 mark_read 8
-- END PROG LOAD LOG --
libbpf: prog 'process_packet': failed to load: -13
libbpf: failed to load object 'src/sheriff-buzz.bpf.o'
```

Fig. 5.5: Verifier error caused by omitting the length check in Fig. 5.4

```
int subnet_state(__u32 src_ip)
{
        /* set up loop callback args */
        struct subnet_loop_ctx ctx = {
                .src_ip = src_ip,
                .type = UNKNOWN
        };

        /* iterate through subnet list */
        bpf_for_each_map_elem(&subnet_list, &subnet_loop_callback, &ctx, 0);
        return ctx.type;
}
```

Fig. 5.6: Iterating over subnet array elements

Return Code	Description
XDP_PASS	Pass the packet on to the network stack.
XDP_DROP	Discard packet. Used for the "block" option for blacklisted sources.
XDP_TX	Retransmit packet on the same interface it came from. Used for the "redirect" option for blacklisted sources.

Table 5.1: XDP return codes used in `sheriff-buzz`

`sheriff-buzz` stores blacklisted and whitelisted subnets in an array map. To check a packet using the subnet criteria, `process_packet` iterates through the array map to determine whether its source IP address belongs to any of the subnets therein. This is needed due to the lack of one-to-one mapping between IP and subnet. Bounded loops are loops within an eBPF program where the verifier can statically prove that they will execute a finite and predictable maximum number of times. In short, while the eBPF verifier does not need to know the exact number of iterations at runtime, it needs to be able to determine an upper bound based on the code logic before the program runs. The `bpf_for_each_map_elem()` helper is used to iterate through the array map (see Fig. 5.6) to satisfy the bounded loop requirement. It takes a callback function to run on each element as a parameter. The return value of the callback function determines whether the helper breaks out of the loop or continues iteration, thus allowing the verifier to determine the loop's upper bound.

`process_packet` stores configuration received from user space in an array map for persistent eBPF storage.

5.3.3 XDP Return Codes

`sheriff-buzz` uses the return codes in Table 5.1 when processing packets. To redirect a packet, it alters its IP headers (modifies the destination IP address and recomputes the checksum) and returns `XDP_TX`. Note that `process_packet` always returns `XDP_PASS` when `sheriff-buzz` is running in dry run mode.

5.4 Uretprobes

The uretprobes attach to user space functions that send configuration, blacklist, and whitelist data in order to update eBPF maps. This mechanism allows for event-based user-to-kernel space communication, as opposed to the high overhead and less efficient polling approach.

`sheriff-buzz` uses the following uretprobe programs:

Program Name	Attaches To	Program Role
`read_ip_rb`	`submit_ip_entry()`	reads IP blacklist/whitelist entries sent from user space and updates the `ip_list` hash map
`read_subnet_rb`	`submit_subnet_entry()`	reads subnet blacklist/whitelist entries sent from user space and updates the `subnet_list` array map
`read_port_rb`	`submit_port_entry()`	reads port blacklist/whitelist entries sent from user space and updates the `port_list` hash map
`read_config_rb`	`submit_config()`	reads configuration sent from user space and updates the `config` array map

Version	Key	Value	Maximum entries per IP
1	• Source IP • Destination Port • TCP Flags	• Timestamps • Packet Count	$2^{16} \times 2^8 = 2^{24}$
2	• Source IP • Destination Port	• Timestamps • Packet Count	2^{16}
3	Source IP	• Timestamps • Port Count • Packet Count • Alert Count • Per-port Packet Counts	1

Table 5.2: Hash table design versions

5.5 Packet Analysis and Scan Detection

The packet processing within the main user space thread is implemented through an eBPF ring buffer callback triggered by each set of packet headers received from `process_packet` whenever it returns `UNKNOWN` for the packet action.

The callback function follows the Fig. 4.2 algorithm to update packet details and detect potential flag- or port-based scan alerts caused by the packet.

5.5.1 Packet Hash Table

Design

Table 5.2 illustrates the design iterations for the hash table. Successive versions decreased the maximum required entries per IP as a strategy to optimise lookup times and reduce memory footprint.

The final version significantly stores the total port count directly within source IP entries. This change makes it so that a single lookup per packet suffices to determine whether a source IP reached the port threshold. This is a dramatic improvement over the previous version, which could require up to 2^{16} lookups—depending on port threshold—per packet.

Implementation

I initially considered using the C standard library `hcreate(3)` to implement the hash table, but decided against it as it does not support deletion or updating of individual entries. This functionality is required for `sheriff-buzz` since entries need to be updated for each packet, and blacklisted sources' entries should be deleted—once communicated to the eBPF component—to conserve memory. I opted for `Glib.HashTable` [32] instead as it provides the aforementioned features.

Fig. 5.7 illustrates the implementation of the packet hash table key and value `structs`. `tcp_ports` and `udp_ports` are dynamically allocated hash tables that store the packet counts per port.

5.5.2 Flag-based alerts

Packets with FIN, NULL, or Xmas flag combinations trigger the packet threshold logic. An alert is triggered whenever the total number of packets sent by the source IP reaches the packet threshold. The code shown in Fig. 5.8 implements this logic.

5.5.3 Port-based alerts

`sheriff-buzz` checks whether the number of ports a source IP has sent packets to has reached the port threshold in order to detect a port scan. Fig. 5.9 shows the code that implements this logic.

```
struct key {
        in_addr_t src_ip;
};

struct value {
        time_t first;
        time_t latest;
        int total_port_count;
        unsigned long total_packet_count;
        int alert_count;
        GHashTable *tcp_ports;
        GHashTable *udp_ports;
};
```

Fig. 5.7: Definition of hash table key and value

```
if (protocol == TCP_PNUM) {
        scan_type = flag_based_scan(&event->tcp_header, types);
        /* check packet threshold */
        if (scan_type) {
                if (val->total_packet_count >= packet_threshold) {
                        report_flag_based_alert(scan_type, current_key, val, address, dst_port);
                        is_alert = true;
                }
        }
}
```

Fig. 5.8: Flag-based scan detection and alert reporting

```
init_entry(packet_table, current_key, val, dst_port, protocol);
/* check port threshold */
if (val->total_port_count >= port_threshold) {
        report_port_based_alert(types.PORT_SCAN, current_key, val, address, port_threshold);
        is_alert = true;
}
```

Fig. 5.9: Port scan detection and alert reporting

```c
void print_stats(int signum)
{
        char buf[MAX_LOG_MSG];

        snprintf(buf, MAX_LOG_MSG, "stats: %ld packets per second, %ld blocked IPs\n",
                        packet_rate(&total_packet_count, &total_handle_time), total_blocked_ips);
        write(LOG_FD, buf, strlen(buf));
}

void setup_signal_handlers()
{
        /* set up SIGUSR1 handler */
        struct sigaction stats_action;

        stats_action.sa_handler = print_stats;
        sigemptyset(&stats_action.sa_mask);
        stats_action.sa_flags = SA_RESTART;

        if (sigaction(SIGUSR1, &stats_action, NULL) == -1) {
                log_error(LOG, "SIGUSR handler: %s\n", strerror(errno));
                exit(errno);
        }
}
```

Fig. 5.10: `SIGUSR1` handler

5.6 Statistics Reporting

To meet Requirement FC3 (Statistics reporting), `sheriff-buzz` provides on-demand statistics. This is triggered when a user sends the `SIGUSR1` signal to the process (e.g. by using `kill -USR1 $(pidof sheriff-buzz)`). In response, `sheriff-buzz`, via a signal handler implemented with `sigaction(2)`, logs a line containing the mean packets per second and the count of IPs blacklisted due to port scan detection.

`print_stats()`—the function that logs the statistics—uses `write(2)` because it is *async-signal-safe* as per the POSIX standard (see `signal-safety(7)`).

5.7 Application Settings

The user can specify options as command-line arguments or via a config file.

5.7.1 Command-Line Arguments

`sheriff-buzz` accepts command-line arguments parsed using the C standard library function `getopt(3)`. Both short and long options are implemented:

```
usage: sheriff-buzz -i <interface> | -a <address> [<args>]
-i, --interface <name>: name of network interface to attach to
-a, --address <address>: address of network interface to attach to
-c, --config <filename>: path to config file
-l, --log <filename>: path to log file
-b, --bpf-obj <path>: path to BPF object file
-s, --skb-mode: enable SKB mode (use if native XDP not supported)
-d, --dry-run: enable dry run mode
-t, --test: enable testing mode
-h, --help: print this message and exit
```

The network interface is the only required argument, specified using `-i` or `-a`. These options are mutually exclusive, and the first one provided takes precedence if both are used.

```
if (poll_num > 0) {
        if (poll_fd.revents & POLLIN) {
                /* inotify events available */
                handle_inotify_events(inotify_fd, CONFIG_PATH, CONFIG_FILENAME,
                                current_config, lock);
        }
}
```

Fig. 5.11: Polling the config file with `inotify`

Option	Set from	Comments
Config file location	command-line arguments	
Log file location	command-line arguments	
eBPF object location	command-line arguments	The eBPF object contains the bytecode that gets loaded into the kernel
SKB mode on/off	command-line arguments/config file	SKB mode is a later kernel hook used as the fallback for network cards that do not support XDP
Dry run mode	command-line arguments/config file	Output alerts but do not drop/redirect any packets

Table 5.3: Additional configuration options added during development

5.7.2 Config File

The user-supplied config file should be human-readable. The config file uses the JSON format for its intuitive structure. Parsing is handled by cJSON [33], a lightweight single-header library.

To allow automatic configuration reloads while `sheriff-buzz` is running, `inotify(7)` is used to monitor the file for modifications. The `handle_inotify_events()` function (see Fig. 5.11) processes the updates.

Synchronisation of the in-memory configuration across threads is achieved using a `pthread_rwlock_t` read-write lock.

5.7.3 Configuration Options

`sheriff-buzz` supports options in addition to those specified in Table 4.2 in order to accommodate a wider range of use cases. For example, config, log, and eBPF object file locations are configurable. See Table 5.3 for a list of additional `sheriff-buzz` options.

5.8 Database Logging

I chose to use PostgreSQL to implement the database since it has a native C library: `libpq` [34]. Database operations are offloaded to a separate worker thread to allow for near-real-time packet analysis performance.

This thread uses the "upsert" logic (`ON CONFLICT DO UPDATE` [35]) for alert entries, updating the record in-place to avoid redundancy.

The database worker thread consumes tasks from a `TAILQ(3)` work queue populated by the main thread. The database thread sleeps via `pthread_cond_wait(3)` when idle and is woken up by the main thread using `pthread_cond_signal(3)` upon task arrival. This avoids busy waiting and greatly reduces CPU usage.

5.9 Log File

`sheriff-buzz` runs as a background process, logging informational messages, alerts, and errors to a log file. A dedicated logging API handles messages based on severity level.

```
db_res = PQexec(conn, query);
err = (PQresultStatus(db_res) != PGRES_COMMAND_OK);
if (err) {
        log_error(LOG, "postgres: %s", PQerrorMessage(conn));
}
```

Fig. 5.12: Logging a PostgreSQL-related error if excecuting `query` fails

```
if (ip_headers && tcp_headers) {
        /* reserve ring buffer sample */
        event = bpf_ringbuf_reserve(&xdp_rb, sizeof(*event), 0);
        if (!event) {
                /* ring buffer allocation failed */
                return;
        }

        /* fill out ring buffer sample */
        event->ip_header = *ip_headers;
        event->tcp_header = *tcp_headers;

        /* submit ring buffer event */
        bpf_ringbuf_submit(event, 0);
}
```

Fig. 5.13: Using the ring buffer `xdp_rb` to send IP and TCP headers to user space

To ensure clear error reporting (Requirement NS2), standard error messages are used where possible, including the C library `strerror(3)` for system errors. For library-specific errors, functions such as `PQerrormessage()` for `libpq` (see Fig. 5.12) and `cJSON_GetErrorPtr()` for cJSON are used to provide detailed error descriptions.

5.10 Inter-Component Communication

This section details how the data is exchanged between `sheriff-buzz` components.

5.10.1 Communication between User and Kernel Space

eBPF has two distinct ring buffer map types for communication between kernel and user space, differing primarily in the direction of data flow:

- `BPF_MAP_TYPE_RINGBUF` (kernel → user): acts as a one-way channel between eBPF programs running in kernel space to send data to applications running in user space. For example, `process_packet` sends packet headers to user space through the `xdp_rb` ring buffer. (see Fig. 5.13) The user space packet analysis thread polls `xdp_rb` using `ring_buffer__poll()` and processes incoming data using a callback function.
- `BPF_MAP_TYPE_USER_RINGBUF` (user → kernel): acts as a one-way channel for applications running in user space to send data to eBPF programs running in kernel space. The user space component submits data (configuration, blacklists, and whitelists) into this type of ring buffer. eBPF uretprobes in kernel space (attached to the user space submission functions) read the data using `bpf_user_ringbuf_drain()` and make it available to other eBPF programs such as `process_packet`.

5.10.2 User Space Component Communication

Section 5.8 details the communication mechanism between the main and database threads in user space using a work queue.

5.11 Documentation

The maintainability of `sheriff-buzz` as a project depends on clear documentation, fulfilling Requirements NM3 and NS1. This section serves as a guide to the documentation provided for both users and developers.

```
usage: sheriff-buzz -i <interface> | -a <address> [<args>]
-i, --interface <name>: name of network interface to attach to
-a, --address <address>: address of network interface to attach to
-c, --config <filename>: path to config file
-l, --log <filename>: path to log file
-b, --bpf-obj <path>: path to BPF object file
-s, --skb-mode: enable SKB mode (use if native XDP not supported)
-d, --dry-run: enable dry run mode
-t, --test: enable testing mode
-h, --help: print this message and exit
```

Fig. 5.14: `sheriff-buzz` help message

```
man(7)                          Miscellaneous Information Manual                          man(7)

NAME
       sheriff-buzz - intrusion detection and response with eBPF

SYNOPSIS
       sheriff-buzz -i <interface> | -a <address> [OPTIONS]

DESCRIPTION
       Detect port scans, blocking/redirecting traffic from hosts deemed suspicious.

       A network interface name or address to attach to is required, all other arguments are optional.

OPTIONS
       --interface, -i <name>
              name of network interface to attach to

       --address, -a <address>
              address of network interface to attach to

       --config, -c <path>
              path to JSON config file

       --log, -l <path>
              path to log file (default is /var/log/sheriff-buzz.log)
```

Fig. 5.15: `man` page excerpt

Specifically, the project `README.md` (Appendix A) lists the dependencies, and Appendix B for an explanation of how the minimum kernel version requirement was determined.

5.11.1 User Documentation

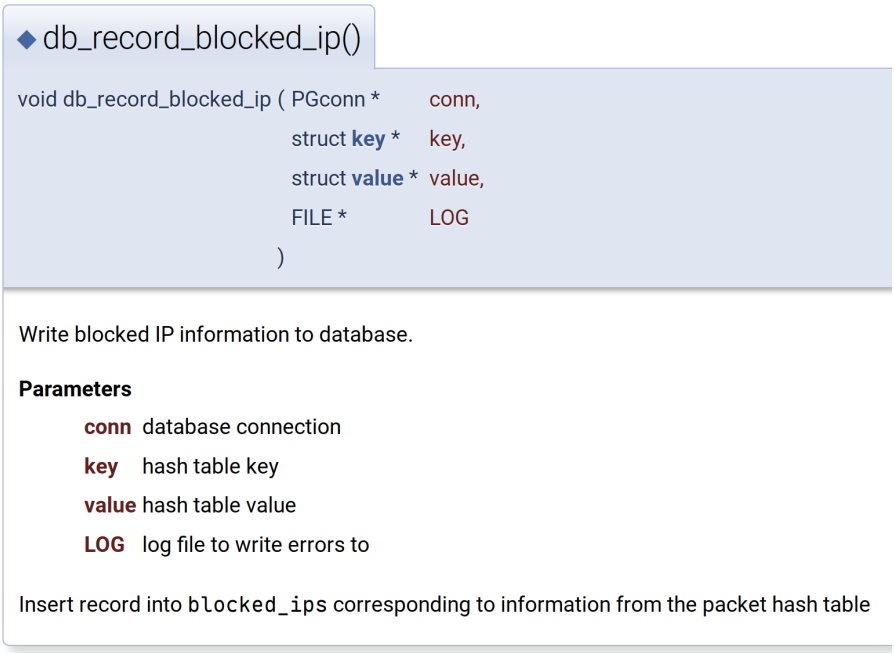
`sheriff-buzz` prints a help message (see Fig. 5.14) when requested with `-h`/`--help` or if arguments are invalid or missing.

It also has a `man` page in `troff` format [36] (see Fig. 5.15).

5.11.2 Developer Documentation

I generated documentation in both HTML and PDF formats by processing doc comments [37] within the source code using the doxygen tool [38]. Fig. 5.16 shows an example of this.

```c
/**
 * @brief Write blocked IP information to database
 * @param conn database connection
 * @param key hash table key
 * @param value hash table value
 * @param LOG log file to write errors to
 * @details Insert record into `blocked_ips` corresponding to information from
 * the packet hash table
 */
void db_record_blocked_ip(PGconn *conn, struct key *key, struct value *value,
                          FILE *LOG)
```



◆ db_record_blocked_ip()

void db_record_blocked_ip (PGconn * conn,
struct key * key,
struct value * value,
FILE * LOG
)

Write blocked IP information to database.

Parameters

conn database connection

key hash table key

value hash table value

LOG log file to write errors to

Insert record into `blocked_ips` corresponding to information from the packet hash table

Fig. 5.16: Doc comment and the corresponding HTML documentation produced with doxygen

Chapter 6  Testing and Performance Analysis

This chapter describes a unit test of `process_packet`, an integration test between `process_packet` and the user space analysis thread, and system testing consisting of real-world traffic on a machine in a DMZ network.

I considered using `hping3` [39] to generate traffic for unit and integration testing, but its hardcoded timeout while waiting for responses[1] from the target IP made it too slow for purpose. I ended up opting for the Python Scapy [40] library since it did not suffer from the `hping3` limitations and is more programmable by design.

6.1 Unit Testing

Unit tests for `process_packet` check the correctness of the return value in different blacklist and whitelist scenarios.

6.1.1 Unit Tests

The following unit tests cover all possible scenarios encountered by `process_packet`:

Test Name	Expected Return Value	Description
block	`XDP_DROP`	cause an IP to get blacklisted, set the `action` to `"block"`, check that subsequent traffic gets dropped
redirect	`XDP_TX`	cause an IP to get blacklisted, set the `action` to `"redirect"`, check that subsequent traffic gets redirected
blacklisted IP before whitelisted subnet	`XDP_DROP`	blacklist an IP that belongs to a whitelisted subnet, check that traffic from such an IP gets dropped
whitelisted IP before blacklisted subnet	`XDP_PASS`	whitelist an IP that belongs to a blacklisted subnet, check that traffic from such an IP gets passed
blacklisted IP before whitelisted port	`XDP_DROP`	blacklist an IP and whitelist a destination port, check that traffic from such an IP to such a port gets dropped

6.1.2 Test Implementation

I added a testing mode to `sheriff-buzz` which can be enabled using the `-t` option on the command line or by setting `"test": true` in the JSON config file. The testing mode implements:

- a test subnet (to be defined in the config file)
- storing of XDP return values for traffic from the test subnet in an eBPF hash map (named `test_results` exposed to user space as a file under the `/sys/fs/bpf` pseudo-filesystem [41]) where the key is the source IP
- logging source IP and destination port for non-dropped traffic from the test subnet (see Fig. 6.1)

[1] `hping3` will not receive any responses since it is faking the source IP; it will have to wait until it times out before sending the next set of packets.

```
pthread_rwlock_rdlock(&config_lock);
if (current_config.test && in_subnet(current_key->src_ip,
                        current_config.test_subnet.network_addr,
                        current_config.test_subnet.mask)) {
        pthread_rwlock_unlock(&config_lock);
        log_info(LOG, "test packet, %s:%d\n", address, dst_port);
} else {
        pthread_rwlock_unlock(&config_lock);
}
```

Fig. 6.1: Logging packets from the test subnet

```
/* definition of test_results map */
struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __type(key, __u32); /* source IP */
        __type(value, __u16); /* XDP return value */
        __uint(max_entries, 65535); /* size of default test subnet 10.10.0.0/16 */;
        __uint(pinning, LIBBPF_PIN_BY_NAME); /* pin to /sys/fs/bpf/test_results */
} test_results SEC(".maps");
...
/* update test_results if testing mode enabled and src_ip belongs to the test subnet */
if (current_config->test &&
        in_subnet(src_ip, current_config->test_network_addr, current_config->test_mask)) {
                bpf_map_update_elem(&test_results, &src_ip, &packet_action, 0);
}
return packet_action;
```

Fig. 6.2: Initialisation and update of unit test results map

The Python unit test script executes the following steps to check `sheriff-buzz` behaviour:

- **Generate Test Input**: create a single random IP address within the test subnet
- **Trigger XDP Program**: send packets to the target using the generated IP as the source address
- **Retrieve XDP Result**: use the Pybpfmaps library [42] to look up the source IP in the `test_results` eBPF hash map and get the resulting `process_packet` XDP return code
- **Validate Outcome**: compare the obtained XDP return code against the expected value to determine if the test passed

6.1.3 Results

`process_packet` passes all the unit tests enumerated in Section 6.1.1. See Fig. 6.3 for a screenshot of running the unit tests.

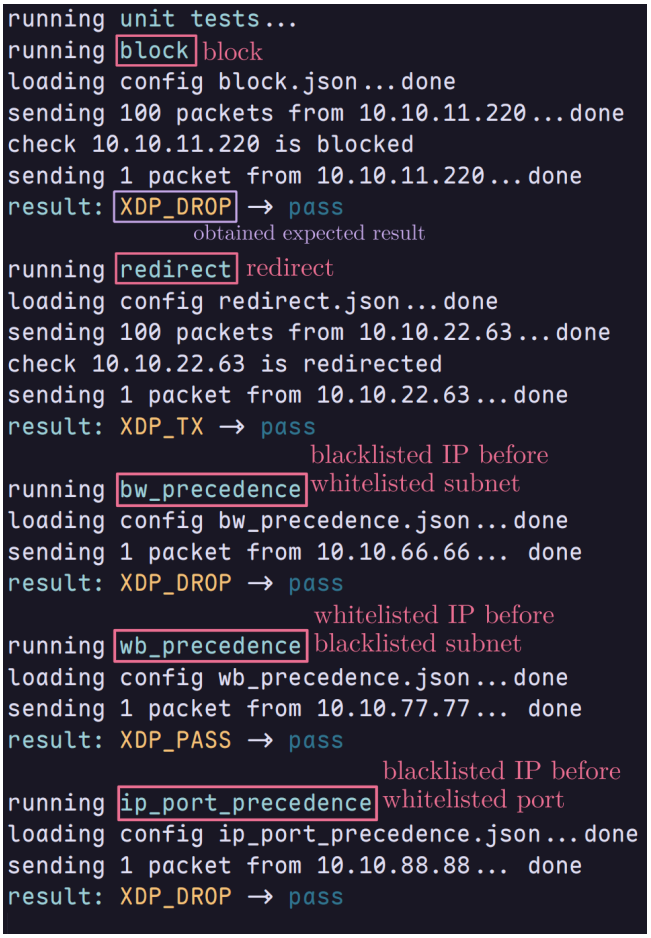


Fig. 6.3: Running `sheriff-buzz` unit tests

```
def lookup(map_name, ip):
    map = bpfmaps.BPF_Map.get_map_by_name(map_name)

    # convert IP string to network-order integer expected by XDP program
    key = int.from_bytes(socket.inet_pton(socket.AF_INET, ip), "little")
    result = map[key]

    return result
...
lookup("test_results", src_ip)
```

Fig. 6.4: Looking up a test result for a given IP in the eBPF map

```
src_ip = packets.rand_src_ip()

for i in range(self.num_tests):
        if not self.fixed_ip:
                src_ip = packets.rand_src_ip()

        # send packet to random port
        dst_port = packets.rand_port()
        packets.gen_packets(src_ip, target, packets.SYN, dst_port)

        result = read_log()
        if compare_packets(result, (src_ip, dst_port)):
                passed += 1
        else:
                failed += 1
```

Fig. 6.5: Integration testing logic

6.2 Integration Testing

Integration tests for the interaction between `process_packet` and the user space analysis thread check that the ring buffer works correctly; packets from random sources and destinations see their headers sent to user space.

6.2.1 Integration Tests

I carried out the following integration tests:

Name	Expected Result	Description
single IP	packets received	send packets a from a single random source IP to random destination ports and check that their headers are received by the user space component
multiple IPs	packets received	send packets from multiple random source IPs to random destination ports and check that their headers are received by the user space component

6.2.2 Test Implementation

The Python integration test script carries out the following steps to check `sheriff-buzz` behaviour:

- **Generate Test Input**: create one or more random IP addresses within the test subnet depending on the test
- **Trigger XDP Program**: send packets to the target from the generated IPs to random destination ports
- **Retrieve Log Output**: use `read_log()` to extract the lines with the `"test packet"` pattern from the log file
- **Validate Outcome**: parse the retrieved lines with `compare_packets()` to verify whether they contain the expected source IP address and destination port combinations

See Fig. 6.5 for a snippet of the implementation. The user can configure the number of test packets the integration test script sends with `-n`; the test script outputs the number of passed and failed packet checks.

6.2.3 Results

`sheriff-buzz` passed several runs of the integration tests. Fig. 6.6 shows an integration test run: 100 packets sent from a single IP followed by 100 packets sent from 100 random IPs.

```
running integration tests...
running fixed_ip single IP
loading config integration.json...done
sending 100 packets from 10.10.17.3...
100%|                                                                                | 100/100 [00:04<00:00, 24.79it/s]
result: 100 total, 100 passed, 0 failed

running rand_ip multiple IPs
loading config integration.json...done
sending 100 packets from 100 random IPs...
100%|                                                                                | 100/100 [00:04<00:00, 24.95it/s]
result: 100 total, 100 passed, 0 failed
```

Fig. 6.6: Running `sheriff-buzz` integration tests

```
#ifdef DEBUG
    void log_debug(FILE *file, char *fmt, ...)
    {
        va_list args;
        va_start(args, fmt);
        log_with_prefix(file, "debug: ", fmt, args);
        va_end(args);
    }
#else
   void log_debug(FILE *file, char *fmt, ...) { }
#endif
```

Fig. 6.7: `sheriff-buzz` debug logging

6.3 System Testing

For system testing, I ran `sheriff-buzz` on a Linux machine with 4 CPUs and 32 GB RAM set up in the DMZ [43] of my home network.

```
$ neofetch os kernel model cpu memory
os: Linux
kernel: 6.12.21-amd64
model: HP EliteDesk 800 G2 SFF
cpu: Intel i5-6500 (4) @ 3.600GHz
memory: 263MiB / 31984MiB
```

With the machine open to the outside world without any protection from the router firewall, `sheriff-buzz` is put in a real-world scenario where it has to detect and respond to port scans as they typically occur on the Internet.

I used the following configuration during testing:

```
{
        "packet_threshold": 1,
        "port_threshold": 10,
        "alert_threshold": 3,
        "whitelist_subnet": ["192.168.1.0/24"]
}
```

Item	Description
`"packet_threshold": 1`	a single FIN, NULL, or Xmas packet is enough to trigger a flag-based alert
`"port_threshold": 10`	source IPs must send packets to 10 different ports in order to trigger a port-based alert
`"alert_threshold": 3`	an IP must cause 3 alerts in order to be blacklisted
`"whitelist_subnet": ["192.168.1.0/24"]`	whitelist home network in order to avoid adding unnecessary analysis overhead from trusted hosts

```
Mon 2025-03-10 00:14:41 debug: local port = 23457 port count = 1     packet count = 1     src IP = 83.222.191.142
Mon 2025-03-10 00:16:08 debug: local port = 4115  port count = 2     packet count = 2     src IP = 83.222.191.142
Mon 2025-03-10 00:21:00 debug: local port = 12317 port count = 3     packet count = 3     src IP = 83.222.191.142
...
Mon 2025-03-10 00:29:37 debug: local port = 30888 port count = 9     packet count = 9     src IP = 83.222.191.142
Mon 2025-03-10 00:33:07 debug: local port = 48051 port count = 10    packet count = 10    src IP = 83.222.191.142
Mon 2025-03-10 00:33:07 alert: port scan (10 or more ports) detected from 83.222.191.142
Mon 2025-03-10 00:33:07 debug: alert count for 83.222.191.142: 1
Mon 2025-03-10 00:33:38 debug: local port = 34067 port count = 11    packet count = 11    src IP = 83.222.191.142
Mon 2025-03-10 00:33:38 alert: port scan (10 or more ports) detected from 83.222.191.142
Mon 2025-03-10 00:33:38 debug: alert count for 83.222.191.142: 2
Mon 2025-03-10 00:38:35 debug: local port = 44789 port count = 12    packet count = 12    src IP = 83.222.191.142
Mon 2025-03-10 00:38:35 alert: port scan (10 or more ports) detected from 83.222.191.142
Mon 2025-03-10 00:38:35 debug: alert count for 83.222.191.142: 3
Mon 2025-03-10 00:38:35 alert: blacklisting 83.222.191.142
```

Fig. 6.8: Log output for traffic from `83.222.191.142`

Source IP	Port scan alerts
83.222.191.142	3
194.180.49.217	2
194.180.49.219	2
208.87.243.205	2

Table 6.1: Summary of `sheriff-buzz` alerts from the 30-minute debug run in the DMZ

6.3.1 Results

30-minute debug run

I compiled `sheriff-buzz` with debug logging enabled (see Fig. 6.7) to track packet and port counts regardless of alerts and verify correct hash map updates and threshold operation.

During a 30-minute run, `sheriff-buzz` analysed traffic from 141 IPs and generated alerts for four IPs. A summary of these alerts is presented in Table 6.1.

Fig. 6.8 is a snippet from the log file leading up to the blacklisting of IP `83.222.191.142`. This IP sent another packet after it was blacklisted, which was dropped by the XDP program:

```
bpf_trace_printk: submitting IP 2394938963, blacklist = 1
bpf_trace_printk: IP lookup: 2394938963 blacklisted
```

`2394938963` is the network order representation of `83.222.191.142`:

```
$ inet_ntop 2394938963
2394938963 -> 83.222.191.142
```

72-hour run

This larger-scale test was conducted without debug logging to avoid impacting performance. Table 6.2 summarises the alerts: 72 IPs triggered alerts, resulting in 54 blacklisted IPs. Example database records for a blacklisted IP are shown in Fig. 6.9.

```
sheriff_logbook=> SELECT * FROM scan_alerts WHERE src_ip = '104.234.115.33';
 id  | dst_tcp_port | alert_type |     src_ip      | packet_count | port_count |        first        |       latest

-----+--------------+------------+-----------------+--------------+------------+---------------------+---------------------
9330 | 799:62163    |          4 | 104.234.115.33  |           11 |         12 | 2025-04-11 19:33:33 | 2025-04-11 19:51:50
sheriff_logbook=> SELECT * FROM blocked_ips WHERE src_ip = '104.234.115.33';
  id  |     src_ip      |        time
------+-----------------+---------------------
 4081 | 104.234.115.33 | 2025-04-11 19:51:50
```

Fig. 6.9: `psql` [44] terminal showing database records for blacklisted IP `104.234.115.33`

Alert Count	Number of IPs
1	12
2	6
3	54

Table 6.2: Summary of `sheriff-buzz` alerts from the 72-hour DMZ run

Name	Type	Maximum number of entries
`ip_list`	LRU hash	16384
`subnet_list`	array	128
`port_list`	LRU hash	1024

Table 6.3: `sheriff-buzz` eBPF map maximum sizes

6.4 Performance Analysis

I analysed the performance of the user space and eBPF components of `sheriff-buzz` in order to evaluate resource utilisation and scalability. This evaluation involved both:

- **Theoretical Analysis**: calculating the time and space complexity of the packet processing algorithms within the kernel and user space
- **Experimental Analysis**: measuring the performance of the user space and eBPF programs

6.4.1 Theoretical Analysis

In this section I analyse the time and space complexity of the main algorithms in `sheriff-buzz`: XDP packet processing and user space analysis.

Time Complexity

Both the kernel and user space components rely heavily on hash tables. The time complexities for hash table operations are:

- **Lookup/Update**: $O(1)$ amortised
- **Iteration** (over n entries): $O(n)$

In the worst-case scenario, the XDP program performs the following lookups for a single packet:

- **Source IP**: $O(1)$ amortised (hash map lookup)
- **Source Subnet**: $O(n)$ (array map iteration where n is the number of configured subnets bounded by the `subnet_list` maximum size specified in Table 6.3)
- **Destination Port**: $O(1)$ amortised (hash map lookup)

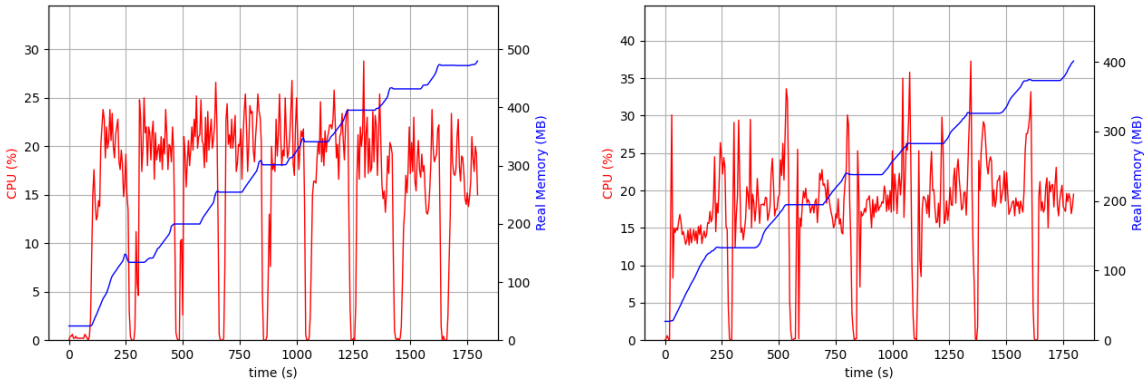
Therefore the worst-case time complexity is $O(n)$ where n is the number of subnets.

The user space component primarily performs hash table lookups and updates, both taking $O(1)$ amortised time. However, when generating a port-based alert, `sheriff-buzz` determines the minimum and maximum port numbers involved before recording the alert. This requires iterating through the offending IP's associated port entries (stored in a hash table). This has a time complexity of $O(n)$ where n is the number of ports the source IP has sent packets to, bounded by the port threshold. In practice, the typical port threshold value is set by the user in the interval $[\![10\,..\,100]\!]$.

Space Complexity

Kernel-space memory usage is constant ($O(1)$) as `sheriff-buzz` utilises fixed-size eBPF maps and its eBPF programs operate within the strict 512-byte eBPF stack size limit.

For user space data (Section 5.5.1 and Fig. 5.7), a primary hash table stores entries for n source IPs ($O(n)$ size). Each entry points to two secondary hash tables (TCP and UDP ports) capped at m (the port threshold) entries. The combined structure leads to a space complexity of $O(n) \times O(m) = O(nm)$. Assuming a fixed, reasonable port threshold m, this behaves effectively as $O(n)$.



(a) bare-metal machine with 4 CPUs and 32GB of memory (b) virtual machine with 2 CPUs and 2GB of memory

Fig. 6.10: `sheriff-buzz` CPU and memory usage over 30 minutes on different machines

6.4.2 Experimental Analysis

CPU and Memory Usage

I used the Python `psrecord` utility [45] to measure `sheriff-buzz`'s CPU and memory usage under heavy load.

The graphs in Fig. 6.10 reflect a 30-minute test with traffic from $\sim$ 1000 IPs (in batches of 100 concurrently at a time) to $\sim$ 16000 distinct ports each. Usage trends are consistent across different hardware configurations, showing similar patterns in resource consumption as depicted in Fig. 6.10(a) and (b).

CPU usage drops to near-zero and memory consumption plateaus in the absence of traffic, reflecting the halt in packet processing and hash table updates. Conversely, memory usage grows with the number of tracked IPs and ports but decreases when IPs are blacklisted as their corresponding hash table entries are removed.

The analysis confirmed `sheriff-buzz`'s viability on resource-constrained machines, demonstrating successful operation on a dual-CPU virtual machine with 2GB of memory (Fig. 6.10(b)), where memory usage peaked at around 400MB during 30 minutes of heavy traffic. However, continuous operation under such a load would necessitate periodic restarts to avoid exhausting memory.

I also assessed resource usage under more realistic conditions during the 72-hour DMZ run (details in Section 6.3.1), with results presented in Fig. 6.11. `sheriff-buzz`'s resource consumption over this extended time frame was minimal, peaking at approximately 0.1% CPU and little over 25.75MB memory. The memory management behaviour was consistent with previous observations (Fig. 6.10), showing memory reclamation upon IP blacklisting.

Flame Graphs

I used Linux `perf(1)` and FlameGraph [46] to visualise the stack traces from the profiled `sheriff-buzz` user space component in order to identify the most frequent code paths. Fig. 6.12(b) zooms in on `handle_event()`: the ring buffer callback. The majority of calls are to hash table lookups and updates (e.g. `init_entry()`), scan detection (e.g. `flag_based_scan()`), and logging (calls to `printf`-related functions), which is expected.

eBPF Program Performance

I used `bpftool` [47] to analyse the performance of `sheriff-buzz`'s eBPF component, with a focus on the frequently run `process_packet`.

`bpftool prog profile` measures eBPF performance metrics. I used it to sample the count of instructions per CPU cycle for the XDP program under heavy network traffic. As shown in Table 6.4, the number of instructions per cycle was consistent across different test machines, suggesting hardware-agnostic execution profile behaviour.

`bpftool prog show` displays information about loaded eBPF programs such as run counts and total runtime in nanoseconds. The latter metric requires eBPF runtime stats which can be enabled via the kernel `sysctl(2)` `kernel.bpf_stats_enabled`. See Fig. 6.13 for example output.

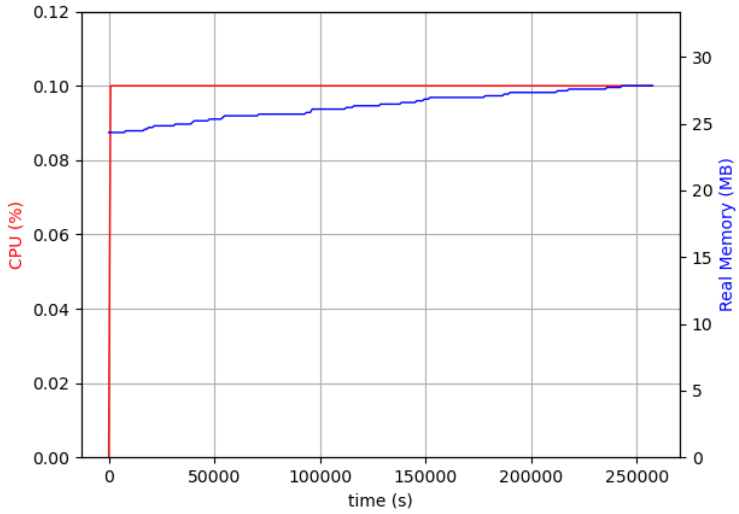


Fig. 6.11: CPU and memory usage on a DMZ host over 72 hours

Machine	Instructions per cycle
Physical (4 CPUs, 32GB RAM)	0.83
Virtual (2 CPUs, 2GB RAM)	1.06

Table 6.4: `bpftool` program profiling on different machines under heavy load

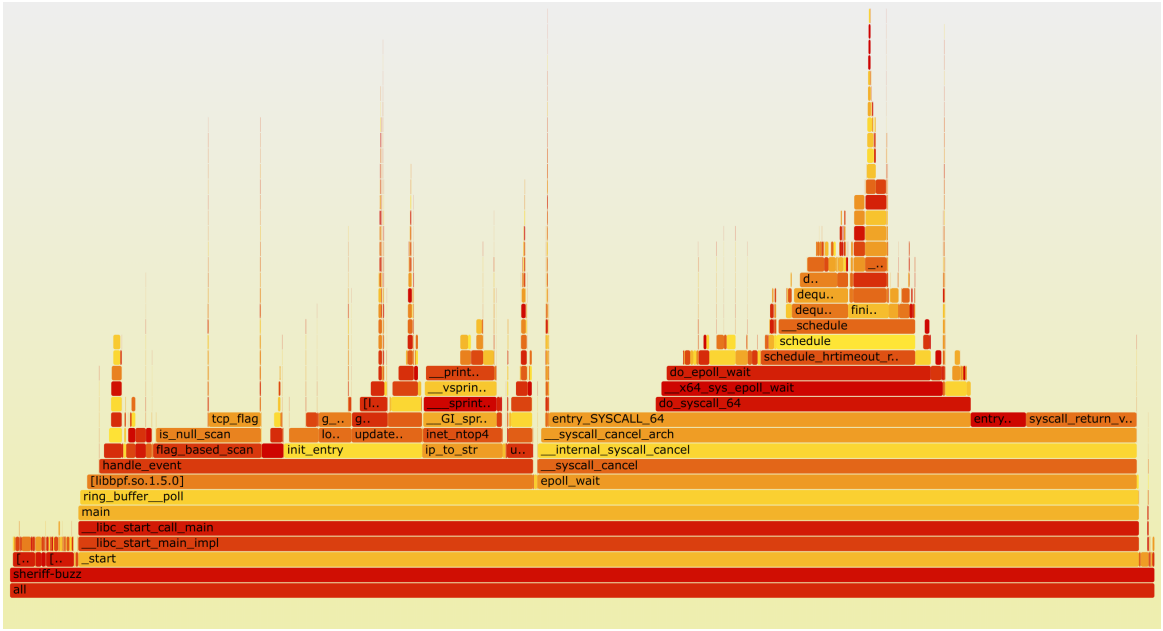
I inferred the number of packets processed per second by the XDP program with:

$$\text{packets per second} = \frac{\texttt{run\_cnt}}{\texttt{run\_time\_ns}} \times 10^9$$

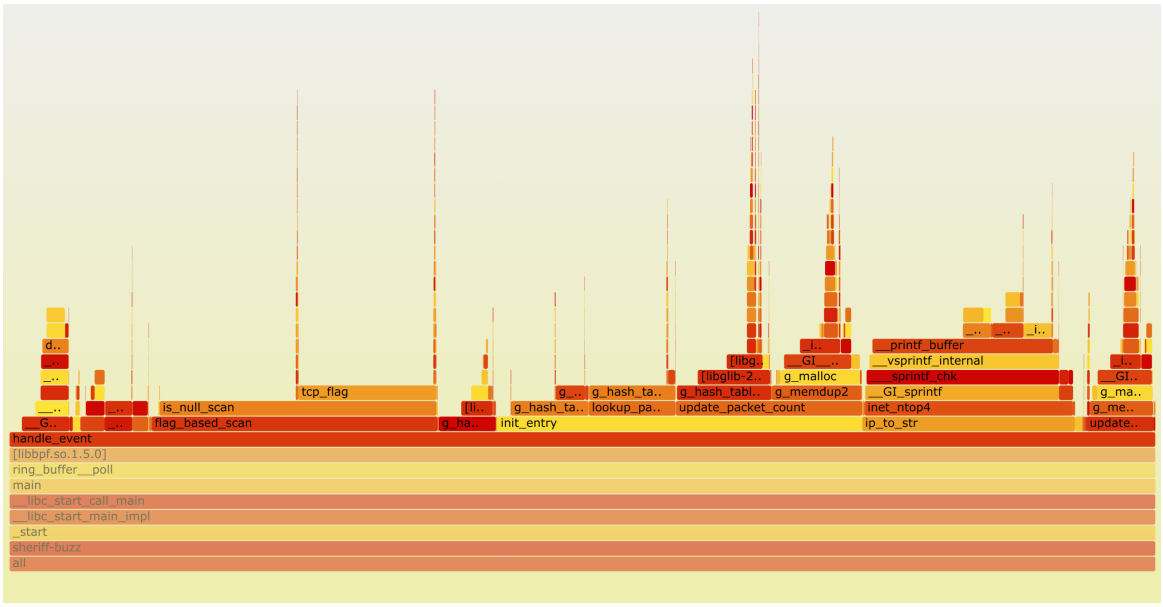
See Table 6.5 for the results. `process_packet` achieved a throughput of over 500,000 packets per second, demonstrating its capacity to handle high-traffic network environments.

Scenario	Packets per second
30-minute heavy load	553179
72-hour DMZ run	293784

Table 6.5: XDP program throughput



(a) Global



(b) Zoomed in on `handle_event()`

Fig. 6.12: Flame graphs for `sheriff-buzz` under heavy load

```
# bpftool prog show name process_packet
7710: xdp  name process_packet  tag 659ccbadff5a1555  gpl run_time_ns 883152085 run_cnt 259456
        loaded_at 2025-04-10T22:55:44+0100  uid 0
        xlated 2560B  jited 1542B  memlock 4096B  map_ids 5248,5240,5241,4495,5242,5243
        btf_id 9138
        pids sheriff-buzz(51956), bpftool(102683)
```

Fig. 6.13: `bpftool prog show` output

Chapter 7  Project Management

This chapter describes my development strategies over the course of the project.

7.1 Timeline

I spent the first semester researching eBPF technology, learning about how it works, its use cases, and its software ecosystem. I started developing `sheriff-buzz` in November as the first of a handful of eBPF case studies. Initially, the aim was to implement a simple flag-based scan detection system. However, starting in late December, `sheriff-buzz` quickly grew into a larger than planned application with several port scan detection mechanisms, multithreaded design, database logging, and flexible configuration. `sheriff-buzz` became the whole project.

I implemented the essential features in December and January. February was spent adding other planned features, improving existing code, and fixing bugs. I spent March preparing for the project demonstration and code refactoring. In April I wrote the project report.

Weekly meetings with my supervisor—supported by detailed reports and demonstrations—facilitated ongoing progress monitoring.

7.2 Development Practices

I used Git [48] to track my project files, using atomic commits to simplify tracking down regressions. I followed a test-driven development methodology to ensure that regressions are caught early.

I systematically used the `clang(1)` static analyser and `valgrind(1)` to hunt for bugs.

I used the note-taking application Obsidian [49] to document my research and keep track of my progress.

I set up a old refurbished HP desktop running Debian Linux as a dedicated development environment. I also made frequent use of virtual machines to test different Linux distributions (Fedora, Ubuntu, and OpenSUSE) and Linux kernel versions.

Chapter 8  Evaluation

This chapter evaluates `sheriff-buzz` against the requirements set out in chapter 3. The "Details" column references the relevant report sections for each requirement.

8.1 Functional Requirements

Functional Must Have (FM)

ID	Name	Status	Details	Comments
FM1	Port scan detection	**Met**	Section 5.5	port scans detected through packet header analysis
FM2	IP blocking	**Met**	Sections 5.3 and 5.10.1	XDP program can be instructed to drop traffic from flagged IPs
FM3	Logging	**Met**	Sections 5.9 and 6.1.1	log levels implemented as well as optional test and debug mode logging
FM4	Alert database storage	**Met**	Sections 4.5 and 5.8	`scan_alerts` table
FM5	Blocked IP database storage	**Met**	Sections 4.5 and 5.8	`blocked_ips` table
FM6	User configuration	**Met**	Section 5.7	config file in JSON format, reloaded when modified

Functional Should Have (FS)

ID	Name	Status	Details	Comments
FS1	IP redirection	**Met**	Sections 5.3 and 5.10.1	XDP program can be instructed to redirect traffic to a configurable IP
FS2	IP blacklist/whitelist	**Met**	Sections 5.7.2 and 5.10.1	`ip_list` eBPF hash map can be updated based on config file
FS3	Subnet blacklist/whitelist	**Met**	Sections 5.7.2 and 5.10.1	`subnet_list` eBPF array map can be updated based on config file

Functional Could Have (FC)

ID	Name	Status	Details	Comments
FC1	Port whitelist	**Met**	Sections 5.7.2 and 5.10.1	`port_list` eBPF hash map can be updated based on config file
FC2	Dry run mode	**Met**	Sections 5.3 and 5.7.3	in dry-run mode, XDP program obeys whitelist rules and sends all other packet headers on for analysis
FC3	Statistics reporting	**Met**	Section 5.6	report mean packets per second and current number of blocked IPs upon receipt of `SIGUSR1`
FC4	Config reload	**Met**	Section 5.7.2	`inotify(7)` worker thread

8.2 Non-functional Requirements

Non-functional Must Have (NM)

ID	Name	Status	Details	Comments
NM1	Scalability	**Met**	Section 6.4.2	copes when concurrently targeted by high number of IPs
NM2	Resource efficiency	**Met**	Sections 5.5.1 and 6.4	optimised packet hash table memory usage, showed low resource usage through performance analysis
NM3	User documentation	**Met**	Section 5.11	usage message, man page, configuration guide in repository
NM4	Database error resilience	**Met**	Section 5.8	failure of the database worker thread is not fatal
NM5	CO-RE	**Met**	Section 5.2 and Appendix B	using `libbpf` [13]

Non-functional Should Have (NS)

ID	Name	Status	Details	Comments
NS1	Developer documentation	**Met**	Section 5.11	doc comments [37] used to generate documentation with doxygen [38]
NS2	Clear error messages	**Met**	Sections 5.9 and 6.3	logging based on standard error descriptions

Non-functional Could Have (NC)

ID	Name	Status	Comments
NC1	Database reconnection	**Not Met**	no attempts are made to reconnect to the database in case of lost connection

8.3 Overall Evaluation

`sheriff-buzz` successfully meets all functional requirements, demonstrating its capability to detect and respond effectively to real-world port scans. It offers robust logging options—to file and database—and user configuration support, catering to a variety of deployment scenarios.

Using eBPF for port scan detection proved to be highly effective. The XDP program achieves high packet processing and allows for rapid decision making thanks to its low overhead. Complementing this, the uretprobes facilitate effective event-based communication by attaching directly to relevant user space functions.

The eBPF component provides significant maintainability benefits over kernel modules or direct kernel source modifications due to its smaller footprint and the abstraction provided by kernel hook points. CO-RE also simplifies deployment by eliminating the need for recompilation across kernel versions.

Implementing the user space component using `pthreads` was demonstrably highly beneficial to the overall performance of the program.

8.3.1 Challenges Overcome

The development process presented several learning opportunities through numerous challenges, some of which are discussed in this section.

A significant hurdle involved learning to decipher eBPF verifier error messages (which are in bytecode- see Figs. B.1 and 5.5 for examples) when developing the XDP component. This led me to better understand the eBPF verifier and be more rigorous with validation through systematic use of—but not only—the `clang(1)` static analyser and `valgrind(1)`.

Another critical task was devising an efficient storage strategy for packet data used in scan detection (see subsection 5.5.1); solving this resulted in substantial performance gains and a decreased memory footprint for the user space component.

Implementing automatic configuration reloading on config file modification led me to learn about `inotify`: something I had never considered during the early stages of development.

I would like to mention several tools that were extremely valuable to the success of the project. Firstly, my trusty editor Neovim [50] and my terminal multiplexer of choice `tmux` [51] made my workflow a breeze. I learned a lot more about Git even though I thought I knew enough at the start of the project, and for that I am both amazed and terrified.

8.3.2 Limitations

Key limitations of the current `sheriff-buzz` system include:

- a threshold-only detection model which may miss repeated failed connection attempts or low-volume scans
- fixed blacklist/whitelist capacities due to eBPF map size limits although LRU eviction somewhat mitigates this shortcoming
- lack of a user-friendly method to remove blacklist/whitelist entries; only manual removal (using `bpftool map delete`) is possible in the current implementation, which does not constitute an acceptable workaround as it does not scale
- the mandatory requirement for `root` privileges to run the application

Chapter 9  Legal, Social, Ethical, and Professional Issues

This chapter outlines the legal, social, ethical, and professional considerations I took into account when developing `sheriff-buzz`.

9.1 Legal Issues

eBPF programs loaded using the `bpf()` system call go through a license check; if they use any functions marked as "GPL-only", they must have a GPL-compatible license configured, otherwise they are rejected [52]. `sheriff-buzz` uses the licence as GPL. Fig. 9.1 shows how this is done in the eBPF component. The user space component is also distributed under the GPL licence.

I also considered the method of response to port scans; the official `nmap` book [16] states:

> Some people propose attacking back by launching exploits or denial of service attacks against the scan source. This is a terrible idea for many reasons. For one, scans are often forged. If the source address is accurate, it may be a previous victim that the attacker is using as a scapegoat. Or the scan may be part of an Internet research survey or come from a legitimate employee or customer. Even if the source address is a computer belonging to an actual attacker, striking back may disrupt innocent systems and routers along the path. It may also be illegal.

`sheriff-buzz` does not make any attempts to attack port scanning hosts back; it only blocks or redirects traffic.

9.2 Social Issues

To my dismay, I realised too late in the development cycle that I probably should not have used terminology such as "blacklist" and "whitelist" due to their possible racist implications. A future version will replace these terms with neutral language e.g. "block list" and "allow list".

9.3 Ethical Issues

False positives can never be entirely eliminated; running `sheriff-buzz` could lead to blacklisting innocent hosts. However, there are a number of methods available to avoid this in certain cases:

- **Thresholding:** There are configurable port, packet, and alert thresholds so that the user can fine-tune the sensitivity of `sheriff-buzz`.
- **Whitelisting:** The user can whitelist trusted hosts to prevent their traffic from being analysed by the scan detection component.

9.4 Professional Issues

I abided by the BCS Code of Conduct [53]: a code of ethics for IT professionals.

```
char LICENSE[] SEC("license") = "GPL";
```

Fig. 9.1: License code required to load eBPF programs into the kernel

Chapter 10  Conclusion and Future Work

I successfully leveraged eBPF technology to develop `sheriff-buzz`: a port scan detection and response system. In this chapter I discuss possible future enhancements.

10.1 Enhancements

The identified limitations for `sheriff-buzz` provide a list of potential enhancements to pursue. In no particular order, the following additions would greatly improve `sheriff-buzz`.

- **Database reconnection**: this would negate the need to restart the application in order to resume database operations assuming the underlying problem that led to database connection loss has been resolved in the meantime. This functionality could be implemented through two mechanisms: periodic reconnection retries and signal handling from user space to provide both automatic and manual methods.
- **LRU user space hash tables**: apply an LRU model (used for the eBPF maps) to user space to ensure that memory usage does not exceed a configurable limit. This could be implemented using a dedicated thread and low and high watermarks: for example, when the table if 85% (the high watermark) full, the thread evicts least recently used entries until the table is 45% (the low watermark) full.
- **Per-CPU eBPF maps**: convert the eBPF maps to per-CPU variants (e.g. `BPF_MAP_TYPE_LRU_PERCPU_HASH` [31]) to improve memory locality, which should greatly reduce cache misses and cache line bouncing.
- **Blacklist ageing**: automatically prune blacklist entries after a configurable period of time.
- **Least privilege model**: there are two possible methods to explore:
  - run as `root` and programmatically drop privileges whenever not needed
  - run as a special user with the minimum required `capabilities(7)`

10.2 Improved Port Scan Detection

The port scan detection engine can be extended to identify the following scan types:

- **Common Port Lists**: packets sent to lists of common ports used by tools such as `nmap`
- **IP Reputation**: similar to Suricata [23], check source IP addresses against a database of ill-reputed IPs
- **Fragmented Headers**: scans where packet headers are split across several packets e.g. `nmap -f`
- **Service and Version Detection**: scans as those carried out by `nmap -sV`. This would require deeper packet analysis.

10.3 Log Format

The logging API could be modified to match the binary output format used by `systemd(1)`; this would allow for manipulation using tools such as `journalctl(1)` and interfacing with applications such as Grafana [54].

10.4 Augmented `sheriff-buzz`

`sheriff-buzz`'s scope can be broadened to detect a wider range of malicious activity:

- **Exploit Attempts**: match traffic to known vulnerabilities in operating systems and services
- **Denial of Service (DoS)/Distributed Denial of Service (DDoS)**: recognise patterns of traffic designed to swamp a network

`sheriff-buzz` could become an intrusion detection component of a larger modular eBPF-based system that encompasses other modules including container performance and health monitoring, system activity monitoring (e.g. storage, CPU, and memory), and security auditing.

Bibliography

[1] What is eBPF? An introduction and Deep Dive into the eBPF Technology. Accessed March 20, 2025. URL: https://ebpf.io/what-is-ebpf/#what-is-ebpf.

[2] Jonathan Corbet. BPF: the universal in-kernel virtual machine. Accessed April 13, 2025. May 2014. URL: https://lwn.net/Articles/599755/.

[3] 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. San Diego, California: USENIX Association, 1993, p. 2. URL: https://dl.acm.org/doi/10.5555/1267303.1267305.

[4] ebpf-for-windows: eBPF implementation that runs on top of Windows. Accessed April 13, 2025. URL: https://github.com/microsoft/ebpf-for-windows.

[5] Iovisor. ubpf: Userspace eBPF VM. Accessed April 15, 2025. URL: https://github.com/iovisor/ubpf.

[6] qmonnet. rbpf: Rust virtual machine and JIT compiler for eBPF programs. Accessed April 15, 2025. URL: https://github.com/qmonnet/rbpf.

[7] Yutaro Hayakawa. "eBPF Implementation for FreeBSD". In: BSDCan2018. Ottawa, Canada, 2018. URL: https://papers.freebsd.org/2018/bsdcan/hayakawa-ebpf_implementation_for_freebsd.

[8] Matt Fleming. A thorough introduction to eBPF. Accessed April 13, 2025. Dec. 2017. URL: https://lwn.net/Articles/740157.

[9] eBPF verifier. Accessed March 19, 2025. URL: https://docs.kernel.org/bpf/verifier.html.

[10] Dominik Scholz et al. "Performance Implications of Packet Filtering with Linux eBPF". In: 2018 30th International Teletraffic Congress (ITC 30). Vol. 01. 2018, pp. 209–217. DOI: 10.1109/ITC30.2018.00039.

[11] Lei Song and Jiaxin Li. "eBPF: Pioneering Kernel Programmability and System Observability - Past, Present, and Future Insights". In: 2024 3rd International Conference on Artificial Intelligence and Computer Information Technology (AICIT). 2024, pp. 1–10. DOI: 10.1109/AICIT62434.2024.10730620.

[12] BPF CO-RE (Compile Once - Run Everywhere). Accessed March 24, 2025. URL: https://docs.kernel.org/bpf/libbpf/libbpf_overview.html#bpf-co-re-compile-once-run-everywhere.

[13] libbpf Documentation. Accessed March 28, 2025. URL: https://docs.kernel.org/bpf/libbpf/index.html#libbpf.

[14] Toke Høiland-Jørgensen et al. "The eXpress data path: fast programmable packet processing in the operating system kernel". In: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66. ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. URL: https://doi.org/10.1145/3281411.3281443.

[15] Robert W. Shirey. Internet Security Glossary, Version 2. RFC 4949. Aug. 2007. DOI: 10.17487/RFC4949. URL: https://www.rfc-editor.org/info/rfc4949.

[16] Gordon Lyon. Nmap network scanning: Official NMAP project guide to network discovery and security scanning. 2008. ISBN: 9780979958717.

[17] Nmap: the Network Mapper. Accessed March 31, 2025. URL: https://nmap.org/.

[18] Snort- Network Intrusion Detection & Prevention System. Accessed March 20, 2025. URL: https://www.snort.org/.

[19] Network Intrusion Detection System Mode. Accessed March 21, 2025. URL: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node6.html.

[20] The Programming Language Lua. Accessed April 7, 2025. URL: https://www.lua.org/about.html.

[21] Configuration - Snort 3 Rule Writing Guide. Accessed April 3, 2025. URL: https://docs.snort.org/start/configuration.

[22] Shie-Yuan Wang and Jen-Chieh Chang. "Design and implementation of an intrusion detection system by using Extended BPF in the Linux kernel". In: Journal of Network and Computer Applications 198 (2022), p. 103283. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103283.
[23] Suricata. Accessed April 15, 2025. URL: https://docs.suricata.io/en/suricata-7.0.9/what-is-suricata.html.
[24] 21. eBPF and XDP — Suricata 7.0.9 documentation. Accessed April 13, 2025. URL: https://docs.suricata.io/en/suricata-7.0.9/capture-hardware/ebpf-xdp.html.
[25] Sebastiano Miano et al. "Securing Linux with a faster and scalable iptables". In: SIGCOMM Comput. Commun. Rev. 49.3 (Nov. 2019), pp. 2–17. ISSN: 0146-4833. DOI: 10.1145/3371927.3371929. URL: https://doi.org/10.1145/3371927.3371929.
[26] Martin A. Brown. Traffic Control HOWTO. Oct. 2006. URL: https://tldp.org/HOWTO/html_single/Traffic-Control-HOWTO/.
[27] Gustavo Bertoli et al. "Evaluation of netfilter and eBPF/XDP to filter TCP flag-based probing attacks". In: Sept. 2020.
[28] João Monteiro and Bruno Sousa. "eBPF Intrusion Detection System with XDP Offload support". In: 2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). 2024, pp. 1–6. DOI: 10.1109/NFV-SDN61811.2024.10807487.
[29] Khadija Sania Ahmad et al. "Fuzzy_MoSCoW: A fuzzy based MoSCoW method for the prioritization of software requirements". In: 2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT). 2017, pp. 433–437. DOI: 10.1109/ICICICT1.2017.8342602.
[30] iovisor. BPF Compiler Collection. Accessed March 28, 2025. URL: https://github.com/iovisor/bcc.
[31] BPF_MAP_TYPE_HASH, with PERCPU and LRU Variants. Accessed April 11, 2025. URL: https://docs.kernel.org/bpf/map_hash.html.
[32] Glib.HashTable. Accessed March 27, 2025. URL: https://docs.gtk.org/glib/struct.HashTable.html.
[33] Dave Gamble. cJSON: Ultralightweight JSON parser in ANSI C. Accessed March 27, 2025. URL: https://github.com/DaveGamble/cJSON.
[34] PostgreSQL: Documentation: 17: Chapter 32. libpq- C Library. Accessed March 27, 2025. URL: https://www.postgresql.org/docs/current/libpq.html.
[35] PostgreSQL: Documentation: 17: INSERT: ON CONFLICT Clause. Accessed April 6, 2025. URL: https://www.postgresql.org/docs/current/sql-insert.html#SQL-ON-CONFLICT.
[36] Ralph Corderoy. troff.org – the Text Processor for Typesetters. Accessed April 12, 2025. URL: https://troff.org/.
[37] Doxygen: Documenting the code. Accessed April 13, 2025. URL: https://www.doxygen.nl/manual/docblocks.html.
[38] Doxygen homepage. Accessed April 12, 2025. URL: https://www.doxygen.nl/index.html.
[39] Hping- Active Network Security Tool. Accessed April 6, 2025. URL: https://www.hping.org/.
[40] Scapy. Accessed April 6, 2025. URL: https://scapy.net/.
[41] Pinning. Accessed April 6, 2025. URL: https://docs.ebpf.io/linux/concepts/pinning/.
[42] Peter Stolz. Pybpfmaps. Accessed April 6, 2025. URL: https://github.com/PeterStolz/pybpfmaps.
[43] What does the DMZ setting on routers do. Accessed March 19, 2025. URL: https://www.microcenter.com/tech_center/article/6780/what-does-the-dmz-setting-on-routers-do.
[44] PostgreSQL: Documentation: 17: psql. Accessed April 12, 2025. URL: https://www.postgresql.org/docs/current/app-psql.html.
[45] astrofrog. psrecord: Record the CPU and memory activity of a process. Accessed April 8, 2025. URL: https://github.com/astrofrog/psrecord.
[46] Brendan Gregg. FlameGraph: Stack trace visualized. Accessed April 9, 2025. URL: https://github.com/brendangregg/FlameGraph.
[47] libbpf. bpftool: Automated upstream mirror for bpftool stand-alone build. Accessed April 8, 2025. URL: https://github.com/libbpf/bpftool.
[48] Git. Accessed March 31, 2025. URL: https://git-scm.com.
[49] Obsidian. Accessed April 15, 2025. URL: https://obsidian.md.
[50] Neovim. Accessed April 15, 2025. URL: https://neovim.io.
[51] tmux. URL: https://github.com/tmux/tmux.
[52] BPF licensing. Accessed March 19, 2025. URL: https://docs.kernel.org/bpf/bpf_licensing.html#using-bpf-programs-in-the-linux-kernel.
[53] BCS Code of Conduct. Accessed March 20, 2025. URL: https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf.
[54] Grafana: The open and composable observability platform. Accessed April 11, 2025. URL: https://grafana.com/.

[55] Git - git-bisect documentation. Accessed April 15, 2025. URL: https://git-scm.com/docs/git-bisect.
[56] Eduard Zingerman. bpf: Allow reads from uninit stack. Feb. 2023. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6715df8d5d24655b9fd368e904028112b54c7de1.

Appendix A  Project Code

`sheriff-buzz` source code is available at https://github.com/salm4ry/sheriff-buzz

Refer to `README.md` for dependencies, compilation, and running instructions.

Appendix B  Minimum Kernel Version

As part of achieving CO-RE, I had to identify the minimum kernel version that `sheriff-buzz` can run on. Running `sheriff-buzz` on an incompatible kernel causes the eBPF programs to be rejected by the verifier (see Fig. B.1).

I used `git bisect` [55] to identify the precise commit that fixes the verifier failure. Fig. B.2 shows the setup of the bisection; I set known commits that were rejected (`old`) and accepted (`new`) by the eBPF verifier.

I built and booted the Linux kernel from source for each commit the bisection landed on in order to test `sheriff-buzz`. I marked commits where the eBPF code was accepted as `new`, and failing commits as `old`. The bisection managed to find the commit in 12 steps as commit `6715df8d5d24` [56]:

```
$ git show 6715df8d5d24
commit 6715df8d5d24655b9fd368e904028112b54c7de1
Author: Eduard Zingerman <eddyz87@gmail.com>
Date:   Sun Feb 19 22:04:26 2023 +0200

    bpf: Allow reads from uninit stack

    This commits updates the following functions to allow reads from
    uninitialized stack locations when env->allow_uninit_stack option is
    enabled:
    - check_stack_read_fixed_off()
    - check_stack_range_initialized(), called from:
      - check_stack_read_var_off()
      - check_helper_mem_access()

    Such change allows to relax logic in stacksafe() to treat STACK_MISC
    and STACK_INVALID in a same way and make the following stack slot
    configurations equivalent:

      |  Cached state    |  Current state   |
      |   stack slot     |   stack slot     |
      |------------------+------------------|
      | STACK_INVALID or | STACK_INVALID or |
      | STACK_MISC       | STACK_SPILL   or |
      |                  | STACK_MISC    or |
      |                  | STACK_ZERO    or |
      |                  | STACK_DYNPTR     |

    This leads to significant verification speed gains (see below).
    ...
```

This commit falls under the tag `v6.3-rc4`:

```
$ git describe --contains 6715df8d5d24
v6.3-rc4~28^2^2~3^2^2~1
```

This makes the minimum required Linux kernel version 6.3 or any kernel that contains a backport of the identified commit.

```
libbpf: prog 'read_ip_rb': BPF program load failed: Permission denied
libbpf: prog 'read_ip_rb': -- BEGIN PROG LOAD LOG --
0: R1=ctx(off=0,imm=0) R10=fp0
; bpf_user_ringbuf_drain(&ip_rb, ip_rb_callback, NULL, 0);
0: (18) r1 = 0xffff8a00c8cad800       ; R1_w=map_ptr(off=0,ks=0,vs=0,imm=0)
2: (18) r2 = 0x6                      ; R2_w=func(off=0,imm=0)
4: (b7) r3 = 0                        ; R3_w=0
5: (b7) r4 = 0                        ; R4_w=0
6: (85) call bpf_user_ringbuf_drain#209
reg type unsupported for arg#0 function ip_rb_callback#92
caller:
R10=fp0
callee:
frame1: R1=dynptr_ptr(off=0,imm=0) R2_w=0 R10=fp0 cb
9: frame1: R1_w=dynptr_ptr(off=0,imm=0) R2_w=0 R10=fp0 cb
; sample = bpf_dynptr_data(dynptr, 0, sizeof(*sample));
...
R2 type=mem expected=fp, pkt, pkt_meta, map_key, map_value
processed 15 insns (limit 1000000) max_states_per_insn 0 total_states 1 peak_states 1 mark_read 1
        -- END PROG LOAD LOG --
libbpf: prog 'read_ip_rb': failed to load: -13
libbpf: failed to load object 'src/sheriff-buzz.bpf.o'
```

Fig. B.1: Verifier error on an incompatible kernel

```
# only test commits in the BPF subsystem
git bisect start -- kernel/bpf

# mark initial old and new commits
git bisect old 270605317366e4535d8d9fc3d9da1ad0fb3c9d45
git bisect new 94e1c70a34523b5e1529e4ec508316acc6a26a2b
```

Fig. B.2: Git bisection setup