DDoS mitigation and anti-scan filtering with eBPF and XDP

Armillotta Michele- 0001139936
De Divitiis Edoardo - 0001146347
Raffaelli Andrea - 0001146287

"This means we can rebuild everything better. What should we rebuild first?"
- eBPF Documentary

Cybersecurity - Academic year 2024-2025

Table of Contents

I. Introduction

Nowadays, the risk of cyber attacks on various information or communication systems is increasing, therefore controlling and monitoring packets travelling from the network to our system is essential for security. If we analyze the Linux operating system there are several tools that allow packet filtering, such as **iptables** and **nftables**, but in this project we try to exploit the novel technology of **eBPF** and **XDP** to increase the performance and flexibility of packet filtering.

We propose two basic implementations for mitigating DDoS and port scanning attacks using eBPF and XDP, which allows us to analyze and possibly make decisions before the packet is handled by the kernel.

II. Motivation

As mentioned previously in this work we focus on:

- **Overhead reduction and performances:** Traffic should be filtered as soon as possible, but with classic tools that work in the kernel, a dropped packet still induces overhead as it allocates memory and goes through processing phases. eBPF with XDP provides a first line of defense *before* the traffic is handled by the kernel.
- **Flexibility:** Of course there are extremely fast solutions in package management like DPDK, but they lack flexibility. XDP is the sweet spot that allows excellent performance but also native support on Linux kernel and without the complete bypass of the network stack (in case of PASS).

III. eBPF and XDP

eBPF (Extended Berkeley Packet Filter) is an advanced Linux kernel technology that allows custom code to be executed directly in the kernel, safely and efficiently, without the need to modify the kernel source code or load external modules. It was originally designed for network packet filtering, but has evolved into a powerful platform for observability, security, and networking.

eBPF allows developers to write small programs in high-level languages like C or abstract languages like eBPF assembly. These programs are then compiled into bytecode and verified by an 'eBPF verifier' to ensure they are safe to run in the kernel. Once verified, the programs are loaded and executed in the context of the

Linux kernel. The verifier prevents infinite loops, unauthorized memory access, and other potential vulnerabilities. eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point. Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.

Today, eBPF is used extensively to drive a wide variety of use cases: Providing high-performance networking and load-balancing in modern data centers and cloud native environments, extracting fine-grained security observability data at low overhead, helping application developers trace applications, providing insights for performance troubleshooting, preventive application and container runtime security enforcement, and much more.

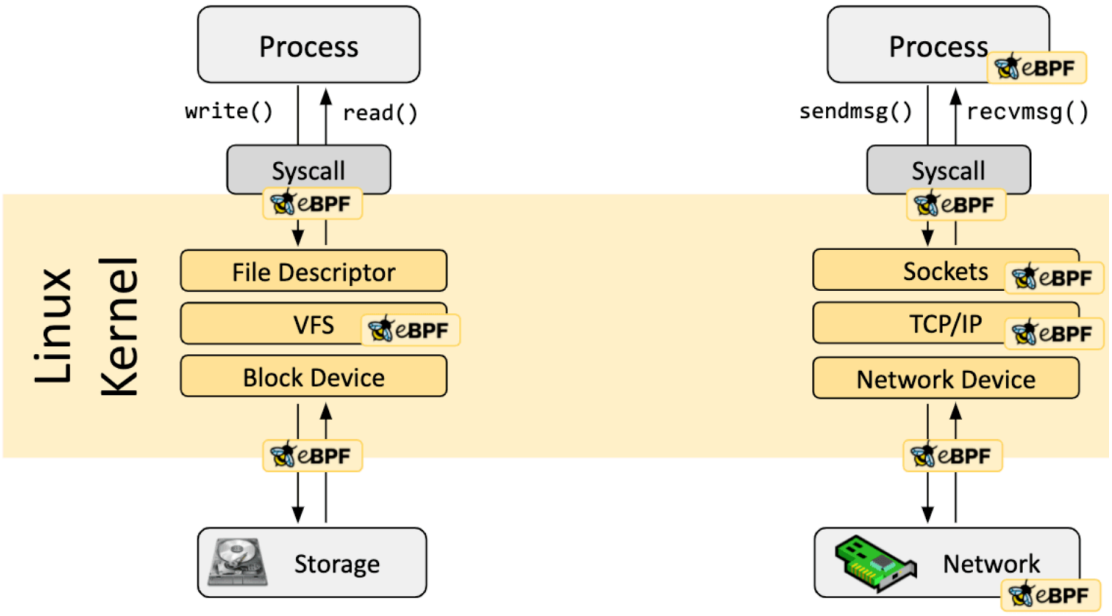

Fig1: eBPF use cases

XDP (eXpress Data Path) is an eBPF-based technology specifically designed for high-performance processing of network packets directly in the network driver, before the packets even reach the traditional kernel networking stack. XDP is extremely efficient and significantly reduces latency and CPU load associated with packet processing.

The idea behind XDP is to add an early hook in the RX path of the kernel, and let a user supplied eBPF program decide the fate of the packet. The hook is placed in the network interface controller (NIC) driver just after the interrupt processing, and before any memory allocation needed by the network stack itself, because memory

allocation can be an expensive operation. Due to this design, XDP can drop 26 million packets per second per core with commodity hardware.

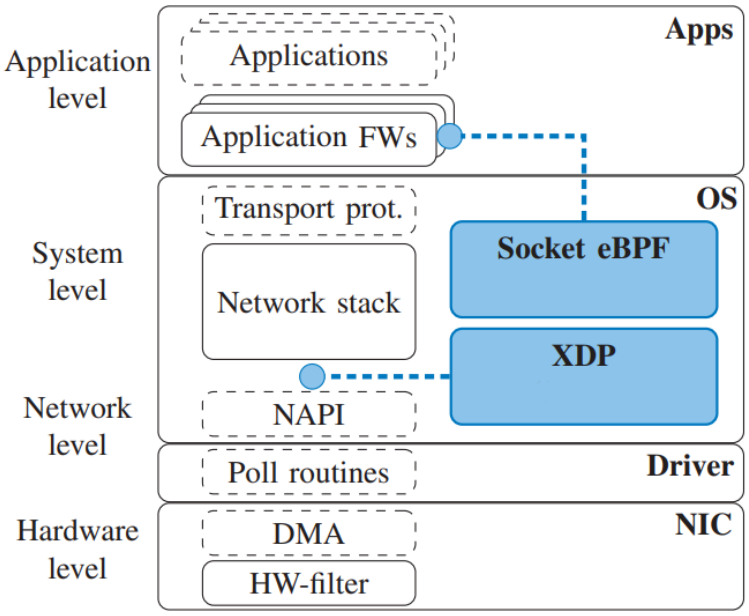

Fig2: XDP placement

IV. (D)DoS and Port Scanning

A **DDoS** ("Distributed Denial of Service") attack is a type of cyber attack that aims to make a service, website, or network resource inaccessible to legitimate users. This is achieved by overloading the target system with a volume of traffic or requests that exceeds its capacity to handle.

The consequences of a DDoS attack can include:

- Service disruption
- Financial losses for businesses and organizations
- Damage to reputation
- Costs associated with recovery and mitigation efforts

In this project we focus on **volumetric DDoS attacks**.

Port scanning is a technique used to identify open, closed, or filtered ports on a network system. Ports represent access points for communications between devices and applications, each associated with a unique identification number (TCP/UDP port). Through this technique, an attacker or system administrator can obtain useful information about the configuration and security of a network device.

When used with malicious intent, port scanning can be a preliminary stage of a cyber attack, such as a DDoS attack, vulnerability exploitation, or system compromise.

V. Systems architecture

a) Setup

The setup used for these two programs and related experiments was using a VM with Linux Debian 11(bullseye), which operates with a network driver *e1000.*
This driver does not have native XDP compatibility and therefore works with *XDP-generic,* which enables XDP program execution for devices without native support at the driver level. In this mode, XDP execution is done by the operating system itself, emulating native execution. This way even devices without explicit XDP support can have programs attached to them, at the cost of reduced performance due to socket buffer allocation and extra steps required to perform the emulation.

There are other modes such as *XDP-native* and *XDP-Offload.* The first way allows running programs directly on the driver, while in the second the eBPF program is offloaded to compatible programmable NICs, achieving even greater performance if compared to the other two modes.

To build these experiments we also use **libbpf bootstrap**, a framework that facilitates the build and use of eBPF technology, it takes into account best practices developed in the BPF community over the last few years and provides a modern and convenient workflow.

b) eBPF maps

The two programs in this project make extensive use of eBPF maps, which are data structures that facilitate the storage and retrieval of data between user space and kernel space. eBPF maps can be accessed not only by eBPF programs but also by user space applications through system calls.

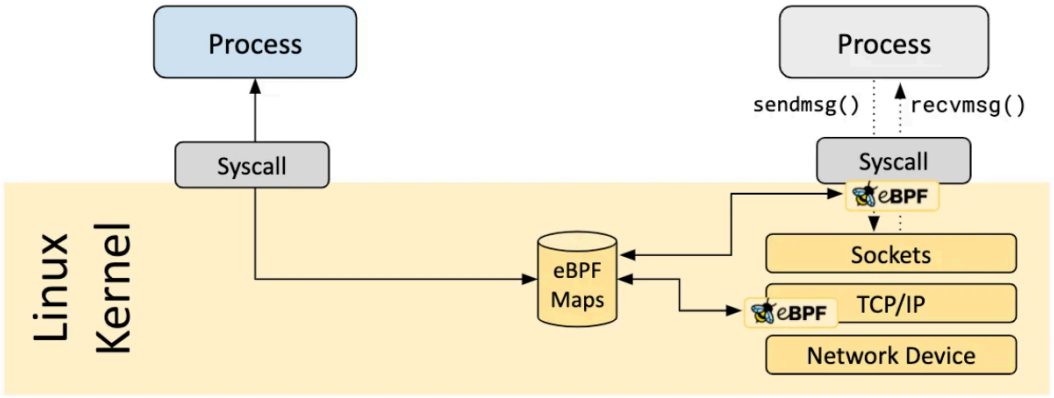

Fig3: eBPF map usage

c) (D)DoS detector architecture and implementation

The DoS detector works by assuming that if one or more IPs send more than a certain number of packets in a second, a DoS attack is in progress. If a DoS attack is detected, only the packets related to that specific IP are dropped, allowing non-malicious connections to continue.

The system is composed of two parts, a user space daemon and a kernel space program (the eBPF program). The first is used to read the user's configuration file and to load the eBPF programs into the kernel. Another important function is the communication and storage of data coming from the eBPF module.

The second, once loaded, is activated each time a packet reaches the network driver. The program analyzes the packet to extract the arrival IP, and checks how many packets that same IP has already sent in the time unit. If the number exceeds the threshold the packets are dropped, otherwise they are forwarded into the network stack.

To allow the eBPF program to keep track of time, a sort of logical clock is used that is updated every second by the userspace daemon. If the times of the two programs do not match, the kernel daemon is updated and the packet count is reset.

Communication between the two components utilizes eBPF maps. Additionally, a ring buffer is employed for logging, which helps offload the event logging process to user space, thereby reducing overhead in kernel space.

Ip	Num. of Packets	Time
127.0.0.1	1000	12
…	…	…

Key	Time
	13

Tab1: If the system detects a mismatch between the “real” time updated by the userspace and the time of the Ip table the neìumber of packets is set to zero and the Ip table time is updated.

d) Anti-scan filtering architecture and implementation

It detects and mitigates port scanning by observing network traffic via the anti-scan filtering system. That is achieved by tracking how many connection attempts are made within a given time by one IP address to different ports. If an IP exceeds the threshold set, it will mark it as a potential scanner and blacklist it.

The system involves mainly two elements: a daemon in user space and a kernel space program by using eBPF. In general, it will be an eBPF program loaded into the kernel and used by this user space daemon for handling communications from user space to kernel space. It does event processing as received from the kernel and performs necessary updates of the blacklist. A kernel space program, deployed as XDP, will inspect incoming packets for IP, port, and protocol, and dispatch the same to user space.

The kernel program verifies whether the source IP has exceeded the connection limit specified in the blacklist map, which is populated by the user space daemon, thereby preventing further scanning attempts. Communication of data between user space and kernel space is facilitated by eBPF maps. A logical clock synchronizes IP access counts between the kernel and user space, ensuring consistency across both components.

VI. Tests and results

a) (D)DoS detector

For the DoS detector a functional test and then multiple DoS attacks were performed. The functional test is very simple: we have 3 UDP servers on the same machine and on different ports, the eBPF program is loaded with relatively low thresholds and then traffic is sent to the 3 servers. Two out of three senders will send legitimate traffic with a rate below the threshold while the third will go above the limit. You can clearly see that all traffic above the limit is blocked for the single connection while the other flows continue undisturbed.

For DDoS attacks, two tools were mainly used: **iperf3** and **hping**. The test is performed by checking the bandwidth of a legitimate 100 Mb/s iperf connection while a DoS attack is mitigated by eBPF. We immediately notice that during the attack the bandwidth loss of the legitimate connection is excessive, many packets are lost even if the DoS detector works without errors. This is probably due to the excessive overhead by the kernel module. The disappointing result obtained can be explained by two elements: excessive use of eBPF maps but above all the use of the XDP-generic mode.

During these tests the average time for a packet to drop when an attack is in progress is 4000 ns, or a rate of 250,000 pps. This result is much lower than what you would expect from an eBPF program, and there are some considerations to make. The tests were performed on a virtual machine with a virtual network card that operates with XDP-generic, this incredibly lowers the performance of an eBPF program. This result, however, even if much lower than the potential of eBPF, still holds its own with the classic filtering tools of the Linux kernel, such as iptables and nftables, although not exceeding their performances.
Probably these same tests performed bare metal with a network card that supports XDP-native would outclass the Linux filtering tools.

Interval	Transfer	Bitrate
3.00-4.00 sec	11.9 MBytes	100 Mbits/sec
4.00-5.00 sec	11.9 MBytes	100 Mbits/sec
5.00-6.00 sec	3.35 MBytes	28.0 Mbits/sec
6.00-7.00 sec	1.10 MBytes	9.21 Mbits/sec

Tab2: we can see the high bandwidth loss at the time (range 5.00-6.00) when the attack starts

b) Anti-scanner

The eBPF-based port scanner demonstrates its ability to detect port scans, even performed by powerful tools like **nmap.** However it introduces a significant bandwidth overhead. This overhead is most evident when comparing the throughput with and without the eBPF program enabled.

These tests were designed to measure the performance of networks, namely bandwidth, latency, and jitter in environments protected by eBPF.

For measurement of these metrics, **iperf3** was used as the primary tool.

1. **Without eBPF:** When the port scanner is run without the eBPF program, the bandwidth remains relatively stable, with a transfer rate around 3.5 Gbits/sec for the entire duration of the test. This indicates that the connection is efficient, with no considerable delays or significant packet loss can be observed, which also shows the usual base performance of the connection one would expect from it.
2. **With eBPF:** Enabling eBPF program significantly reduces throughput (dropping transfer rate to around 1 Gbit/sec) due to additional processing of each packet. Although this impacts bandwidth, connection remains stable with minimal packet loss, jitter and latency, the eBPF program's role in detecting port scans introduces substantial bandwidth limitations.

This table shows the network performance results when connecting to a virtual machine with the eBPF anti-scanner disabled. During the test, the client (host) transmits data to the server (VM) over a duration of 10 seconds using the command:

```
iperf3 --client 192.168.56.10 --port 5000 -t 10
```

The measuring include the total amount of data transferred and the achieved bitrate:

Name	Interval	Transfer	Bitrate
Host (sender)	0.00-10.00 sec	4.14 GBytes	3.55 Gbits/sec
VM without BPF (receiver)	0.00-10.00 sec	4.14 GBytes	3.55 Gbits/sec

Tab3: performance without BPF anti-scan

Network performance results when connecting to a VM with the eBPF anti-scanner enabled:

Name	Interval	Transfer	Bitrate
Host (sender)	0.00-10.00 sec	1.20 GBytes	1.03 Gbits/sec
VM with BPF (receiver)	0.00-10.00 sec	1.19 GBytes	1.02 Gbits/sec

Tab4: performance with BPF anti-scan

To further assess the impact of the eBPF program, a bandwidth-limiting test was done on 1 Gbit/sec by using UDP packets. The results clearly highlighted that when even the bandwidth is throttled back to 1 Gbit/sec, the stability of the connection stayed overall relatively fine and had only negligible jitter while offering acceptable packet loss.

1. **Without eBPF**, the connection showed very stable performance: very low jitter of between 0.003 ms and 0.024 ms, and very low packet loss. The system was able to handle full 1 Gbit/sec bandwidth without introducing any noticeable delays.
2. **With eBPF enabled**, the jitter remained low, between 0.005 ms and 0.05 ms, while packet loss was higher because of the limited bandwidth and overhead introduced by the port scan detection. Despite this, the overall connection remained functional with little impact on the service performance.

Updated network performance (1 Gbit/sec bandwidth limit) with eBPF program active on the receiving VM:

Name	Interval	Transfer	Bitrate	Jitter	Lost/Total Datagrams
Host (sender)	0.00-10.00 sec	913 MBytes	762 Mbits/sec	0.005 ms	202357/863243 (**23%**)
VM (receiver)	0.00-10.00 sec	1.12 GBytes	958 Gbits/sec	0.013 ms	32205/863190 (**3.7%**)

Tab5: performance without BPF anti-scan and bandwidth limit

Therefore, from the results, it can be observed that the eBPF program effectively detects port scans but at the cost of significant bandwidth overhead, which can only allow a transfer rate of 1 Gbit/sec. Although stable, it may not be suitable for high-bandwidth environments unless a balance between security and performance is acceptable. This is more suitable where moderate bandwidth is preferred to be set up with higher transfer speeds.

It should be noted that these results, like the previous ones, were obtained in a virtualized environment that **does not support XDP-native**. The same tests with this feature could reveal much better performance.

VII. Conclusions and future developments

In this project we have seen first-hand the potential of a tool like eBPF, in a complex field like cybersecurity. We have obtained good results in terms of **flexibility** and mediocre results regarding **efficiency**, but much of what has been done can be incredibly improved.

a) (D)DoS detector future developments

There are huge improvement prospects for the dos detector. First of all it would be interesting to perform bare metal tests with XDP-native, to observe the real potential of eBPF.
In addition to this some improvements to the program logic to avoid security holes could be made:

1. Add a global threshold mechanism to avoid that a (huge) set of different IPs saturates the network even if each specific IP remains under the threshold.
2. Implement a blacklist for IPs that go for too long above the threshold to avoid receiving malicious packets
3. Implement a whitelist of trusted IPs

4. Implement dynamic management of the number of controllable IPs to make the solution scalable.

b) Anti scanner future developments

Also for the port scanner it would be interesting to do the same tests using XDP-native in a bare metal environment, and observe if the overhead actually reduces, representing more realistic metrics. For future improvements, we should look into smarter ways to spot scanning patterns. Also, adding a mechanism to limit the rate based on the scanner's behavior would boost its performance. Ensuring the tool can handle bigger networks would make it more flexible fitting for a broader range of setups. With these improvements the tool could become significantly more efficient and exploitable in real-life situations.

VIII.Bibliography

[1] eBPF. eBPF Documentation. https://ebpf.io/

[2] libbof-bootstrap. https://github.com/libbpf/libbpf-bootstrap

[3] Marcelo Abranches, Oliver Michel, Eric Keller, Stefan Schmid. (2021). Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 10.1109/NFV-SDN53031.2021.9665095

[4] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (February 2020), 36 pages. https://doi.org/10.1145/3371038

[5] P. Blazek, T. Gerlich, Z. Martinasek and J. Frolka, "Comparison of Linux Filtering Tools for Mitigation of DDoS Attacks," *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*, Athens, Greece, 2018, pp. 1-5, doi: 10.1109/TSP.2018.8441309.

[6] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak and G. Carle, "Performance Implications of Packet Filtering with Linux eBPF," *2018 30th International Teletraffic Congress (ITC 30)*, Vienna, Austria, 2018, pp. 209-217, doi: 10.1109/ITC30.2018.00039.

[7] D. Melkov, A. Šaltis and Š. Paulikas, "Performance Testing of Linux Firewalls," *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, Vilnius, Lithuania, 2020, pp. 1-4, doi: 10.1109/eStream50540.2020.9108868.

[8] xdp-paper. https://github.com/tohojo/xdp-paper

[9] iperf. https://iperf.fr/

[10] nmap. https://nmap.org/