Accelerating Linux Security with eBPF iptables

Matteo Bertrone, Sebastiano Miano, Fulvio Risso, Massimo Tumolo
Department of Control and Computer Engineering, Politecnico di Torino, Italy

CCS CONCEPTS

• **Networks** → **Firewalls**; *Programmable networks*; *Packet classification*;

KEYWORDS

eBPF, iptables, Linux, XDP

1 INTRODUCTION

Nowadays, the traditional security features of a Linux system are centered on `iptables`, which supports different security policies (e.g., inspect, modify, redirect, drop) to be applied to the traffic received/sent by local applications or to filter the traffic forwarded by the system. Although iptables has been around for 20+ years, it is still the mostly used packet filtering mechanism in the Linux kernel. However, the increase in network speed and the transformation of the type of applications running in a Linux server has led to the consciousness that the current implementation may not be able to cope with the modern requirements particularly in terms of scalability, as the number of rules is dramatically increasing [2].

In recent years, the extended BPF (eBPF) subsystem has been added to the Linux kernel, offering the possibility to execute (almost) arbitrary code when a packet is received or sent, including stateful processing. Notably, this does not require any additional kernel module and offers the possibility to compile and inject this code dynamically, hence facilitating over-the-air updates. The above characteristics make eBPF a perfect candidate to build an iptables clone such as [1], which can be considered more an initial proof-of-concept that filters traffic based on IP addresses than a full iptables replacement. This paper starts from the above activities and it presents a first eBPF-based prototype, `iov-iptables`, which emulates the iptables filtering semantic and exploits a more efficient matching algorithm. Finally, we evaluate our prototype comparing it with the current implementation of iptables, showing how this allows obtaining a notable advantage in terms of performance particularly when a high number of rules is involved, without requiring custom kernels or invasive software frameworks (e.g., DPDK) that could not be allowed in some scenarios (e.g., servers in large datacenters).

2 IOV-IPTABLES DESIGN

The design of `iov-iptables` includes two orthogonal aspects: (*i*) the strategy adopted to preserve the semantic of the iptables firewall policies, and (*ii*) the data plane architecture, which is driven by the algorithm used for packet matching. We leave the detailed techniques and implementations to a future paper and focus here on the key design of the `iov-iptables` architecture.

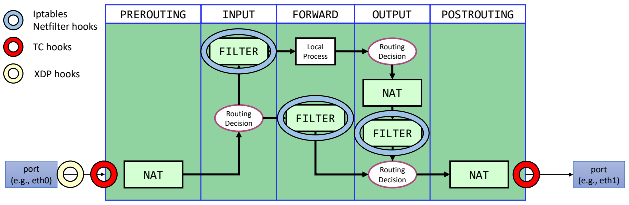


Figure 1: Netfilter vs eBPF hooks

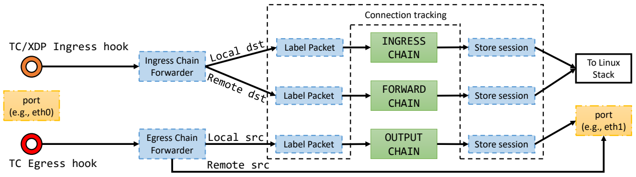
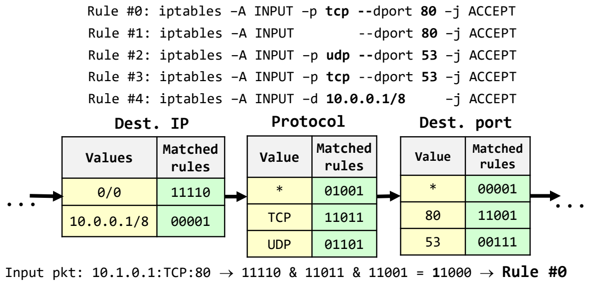


Figure 2: Overall data plane structure

Iptables filtering semantics. Iptables filters packets either in the `INPUT`, `FORWARD` and `OUTPUT` chains of the Linux Netfilter framework, which are located in a different position compared to eBPF hooks (Figure 1). In particular, the XDP hook, available only for incoming traffic, is located at the earliest point in the networking stack. Instead, the Traffic Control (TC) hook, available for both incoming and outgoing traffic, is executed either before the `PREROUTING` or after the `POSTROUTING` hook.The different position of the filtering hooks in Netfilter and eBPF poses non-negligible challenges in preserving the semantic of the iptables rules, which, when enforced in an eBPF program, operate on a different set of traffic compared to the one that would cross the chain they are attached to. As an example, a rule “iptables -A INPUT -j DROP” drops all the incoming traffic directed to the current host, but it does not affect the traffic that is being forwarded by the host itself. A similar “drop all” rule, applied in the ingress XDP/TC eBPF hook will instead drop all the incoming traffic, also the one that would be forwarded by the host itself. `iov-iptables` adds an XDP/TC *Ingress Chain Forwarder* and a TC *Egress Chain Forwarder* eBPF program (Figure 2) to recognize the actual path of each packet and emulate the iptables filtering behavior, redirecting the packet to the proper filtering (`INPUT` or `FORWARD`) chain.



```
Rule #0: iptables -A INPUT -p tcp --dport 80 -j ACCEPT
Rule #1: iptables -A INPUT        --dport 80 -j ACCEPT
Rule #2: iptables -A INPUT -p udp --dport 53 -j ACCEPT
Rule #3: iptables -A INPUT -p tcp --dport 53 -j ACCEPT
Rule #4: iptables -A INPUT -d 10.0.0.1/8     -j ACCEPT
```

Dest. IP

Values	Matched rules
0/0	11110
10.0.0.1/8	00001

Protocol

Value	Matched rules
*	01001
TCP	11011
UDP	01101

Dest. port

Value	Matched rules
*	00001
80	11001
53	00111

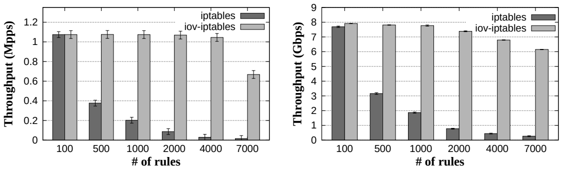
```
Input pkt: 10.1.0.1:TCP:80 → 11110 & 11011 & 11001 = 11000 → Rule #0
```

Figure 3: Example of the LBVS classification pipeline

Matching algorithm. Improving the existing linear search of `iptables` does not appear so difficult. However, many of the existing matching algorithms (e.g., cross-product, decision-tree approaches) require either sophisticated data structures that are not available for eBPF programs [4] or an unbounded amount of memory, which is not desirable for a kernel program. `iov-iptables` uses the Linear Bit Vector Search [3], which proved to be reasonably fast while being feasible with current Linux kernels (and available eBPF maps).

The algorithm consists in creating a specific bi-dimensional table for each field on which packets may match, such as IP addresses, TCP/UDP ports, and more. Each table contains the list of unique values for that field present in the given ruleset, plus a wildcard. Each value in the table is associated to a bitvector of length N, equal to the number of rules, which keeps the list of rules matching that field. Finally, a bitwise `AND` of the intermediate bitvectors returns the list of matched rules; the one with the highest priority (corresponding to the first rule with 1 in the bitvector) is returned. Each map can be implemented in a different way, based on the field characteristics (e.g., longest prefix match in case of IP addresses and ranges; hash tables for TCP/UDP ports). Per-CPU maps are used whenever possible to avoid cache pollution among different CPU cores and increase the effectiveness of parallel processing of multiple packets on different CPU cores. Thanks to the dynamic code injection of eBPF, we created a matching pipeline that contains the minimum number of processing blocks required to handle exactly the fields required by the current ruleset, avoiding unnecessary processing for unused fields. New processing blocks can be added at runtime if the matching against a new field is required, always keeping the optimal number of programs.

Data plane architecture. Figure 2 shows the overall data plane architecture of `iov-iptables`. When a packet reaches a given port, it triggers the *Ingress Chain Forwarder* program attached to the TC or XDP hook; depending on whether the packet is directed to a local application, it jumps to either the `INGRESS` or `FORWARD` chain, which implement the corresponding matching pipeline. Before entering the respective chains, the packet passes from a first *connection tracking* module (implemented as an additional eBPF program), which associates a state to each packet, so that all subsequent chain rules can be correctly applied. When the packet leaves the chains, as result of an `ALLOW` action, it passes through a second conntrack module that saves the state of the session in its internal eBPF map; depending on the chain, the packet is then delivered to the Linux stack or forwarded to the output port. On the other side, when a local application sends a packet out to a netdevice, it will trigger the *Egress Chain Forwarder* program attached to the TC egress hook that, looking at the source IP of the packet decides to forward it directly to the output port (because the `FORWARD` chain has already processed it) or to jump to the `OUTPUT` chain where the classification algorithm will be applied.



(a) UDP throughput (forward) (b) TCP throughput (input)

Figure 4: iov-/iptables performance comparison

3 EVALUATION

We evaluated `iov-iptables` by attaching it to both the TC ingress and egress hook of the host interfaces and compared its performance against `iptables` in two different cases. In the first test, shown in Figure 4(a), we added an increasing number of rules to the `FORWARD` chain of the firewall and we generated a unidirectional stream of 64B UDP packets. In the second, shown in Figure 4(b), we added an increasing number of rules to the `INGRESS` chain to protect locally running applications, and then we calculated the resulting TCP throughput. In both tests, we generated traffic so that only one CPU core is involved in the processing. Results confirm that `iov-iptables` outperforms `iptables` by an order of magnitude when a high number of rules is used, thanks to its improved algorithm and the different optimizations on the classification pipeline that are allowed by the dynamic code injection of eBPF, with a vanilla Linux kernel.

REFERENCES

[1] D. Borkmann. 2018. net: add bpfilter. (feb 2018). https://lwn.net/Articles/747504/

[2] T. Graf. 2018. Why is the kernel community replacing iptables with BPF? (apr 2018). https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables

[3] T.V. Lakshman and D. Stiliadis. 1998. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM Computer Communication Review*, Vol. 28. ACM, 203–214.

[4] S. Miano, M. Bertrone, F. Risso, M. Vásquez Bernal, and M. Tumolo. 2018. Creating Complex Network Service with eBPF: Experience and Lessons Learned. In *High Performance Switching and Routing (HPSR)*. IEEE.

TECHNICAL REQUIREMENTS

- Equipment to be used for the demo:
  - 2 computer screens with VGA or display port sockets
  - 1 PC running the prototype
  - 1 PC acting as traffic generator/sink
  - 2 10Gbps Ethernet links
- Space needed: a table of at least 1 square meter.
- Setup time required: about 30 minutes for the first setup; less than one minute to reset and restart the demo.
- Additional facilities needed: an electrical socket.

DEMONSTRATION STEPS

The demo will show:

- the different throughput of iptables and `iov-iptables` with the same ruleset, with the same input traffic pattern;
- the compatibility in terms of command line among the two tools by using exactly the same command for both;
- the structure of the `iov-iptables` matching pipeline, with different eBPF programs in cascade, each one in charge of the matching on a different field;
- the capability to dynamically update the pipeline with new elementary matching blocks, i.e., in case a rule operating on a new field is added (or vice versa, removing an elementary processing block when the matching on a given field is no longer required).