Toward an eBPF-based clone of iptables

Matteo Bertrone, Sebastiano Miano, Jianwen Pi, Fulvio Risso, Massimo Tumolo

Abstract

Iptables, which is currently the most common firewall on Linux, has shown several limitations over the years, with scalability as a big concern. This paper reports the first results of a project that aims at creating a (partial) clone of iptables, using the eBPF/XDP technology. This project assumes unmodified Linux kernel and guarantees the full compatibility (in terms of semantics and synxtax) with current iptables.

Keywords

eBPF, iptables, netfilter, firewall.

Introduction

Many Linux servers exploit `iptables`, which is part of the `netfilter` [5] kernel subsystem, to protect the server from threats coming from the external network. Although widely used, iptables has been criticized in many aspects, such as for its antiquate matching algorithm (linear search); its syntax, not always intuitive; its old code base, which is difficult to understand and maintain.

Over the years, this triggered the creation of several alternative firewall projects trying to address some of the above mentioned limitations. For example, `ufw` [6] focused on a simpler user interface, although the components behind the hood are still the one used by iptables. Instead, `nftables` [2] proposed an extensible virtual machine that interprets code dynamically generated and loaded from user space, simplifying the kernel source code base and facilitating the possibility to add new features or support new protocols. However, the foregoing projects failed so far to replace iptables in real world deployment, hence leaving it as one of the most used software nowadays.

Recent activities in the Linux networking community are currently investigating whether iptables can be replaced with an eBPF-based clone, which led to the creation of the bpfilter [1] prototype. So far, this work has been focusing on the performance side, showing the advantages of intercepting (hence filtering) packets early in the kernel, even on the smartNICs, and on the user interaction, i.e., how to intercept iptables firewall rules for their emulation in the eBPF framework.

This paper starts from the above activities and presents an eBPF-based (partial) clone of iptables, called `bpf-iptables`, which emulates the iptables filtering semantic and exploits a more efficient matching algorithm.

It presents four additional challenges that need to be tackled in order to obtain a fully-compatible clone of iptables, such as (*i*) how to preserve the semantic of iptables rules when operating with the eBPF hooks; (*ii*) how to choose and implement a fast matching algorithm in eBPF; (*iii*) how to support rules based on connection tracking; and finally (*iv*) how to guarantee the compatibility with the iptables syntax.

A positive answer to the above challenges will enable bpf-iptables to accept vanilla iptables commands without letting a normal user to notice any difference. Iptables rules will be emulated through the proper set of eBPF programs, hence leading to a cleaner and more future proof architecture, and a (possible) increase in processing speed 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).

Prototypal architecture

This Section presents the architecture of the bpf-iptables prototype, derived from the necessity to solve the four main challenges listed in the Introduction.

Preserving iptables semantic

Iptables enables to filter traffic in three different locations, which are called `INPUT`, `FORWARD` and `OUTPUT` chains, as defined by the netfilter framework and shown in Figure 1. As suggested by the names, the first chain applies to traffic that is *terminated* on the host itself; the second can handle traffic that *traverses* the host (e.g., when Linux is asked to act as a router and forward IP traffic between multiple interfaces), while the third operates on traffic *exiting* from the host and directed to the Internet. It is important to notice that the `FORWARD` chain is used also when the traffic traverses the Linux kernel coming from (or directed to) non-root network namespaces, which is becoming a common case in many virtualized deployment (e.g., Kubernetes).

On the other hand, eBPF hook points are different and are located before the traffic control (`TC`) module, which is earlier than the above filtering points for incoming traffic, and later for outgoing traffic, as shown in Figure 1. The different position of the filtering hooks in netfilter and eBPF poses

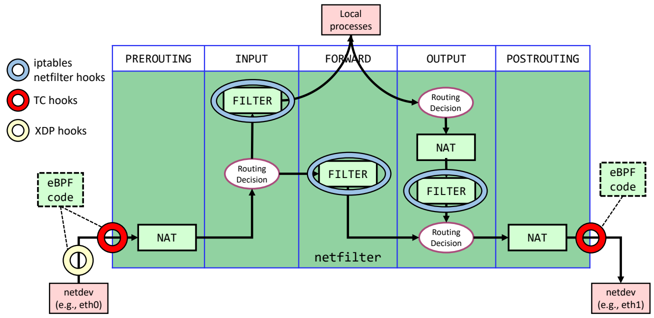


Figure 1: Comparing the location of netfilter and eBPF hooks.

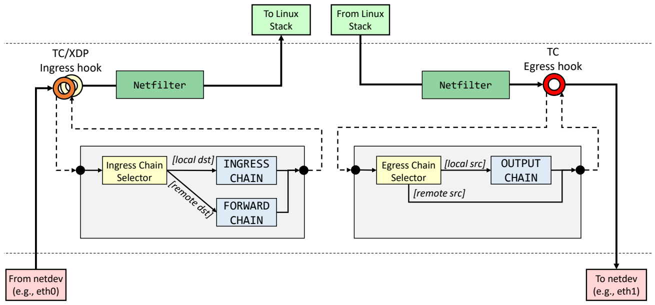


Figure 2: Main filtering architecture of bpf-iptables (without connection tracking).

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, 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 eBPF TC_INGRESS hook point, will instead drop all the incoming traffic, also including the one that would be forwarded by the host itself.

This behavior suggests the necessity to introduce a classification logic in the eBPF hook points that predicts the set of traffic that would reach each individual iptables chain. This would enable bpf-iptables to emulate the same behavior of iptables although operating in a different hook point. A possible solution is the architecture depicted in Figure 2, which features an initial Chain Selector module in charge of predicting which path would be taken by the traffic, followed by the actual filtering block that is configured with exactly the same rules operating in the original iptables chain.

According to this architecture, TC/XDP ingress hooks are used to emulate the filtering behavior of the iptables `INPUT` and `FORWARD` chains. Instead, the `TC_EGRESS` hook emulates only the `OUTPUT` chain; the associated Chain Selector module detects the traffic forwarded by the host and sends it directly in output, since that traffic has already been filtered by `FORWARD` emulation module attached to the TC/XDP ingress hooks.

The Chain Selector block is a simple filter that classifies traffic based on the IP address of the traversed packets, particularly based on the Destination IP address for the Ingress Chain Selector, and Source IP address in case of the Egress Chain Selector. The idea is that traffic would cross the `INPUT` chain only if it is directed to a local IP address, visible from the host root namespace; similarly, a packet would traverse the `OUTPUT` chain only if has been generated locally. This is achieved in our prototype with an additional control logic that (*i*) lists all the IP addresses visible from the root namespace and configures them in the Chain Selector when the systems starts; and (*ii*) attaches to the appropriate NETLINK messages in order to detect any change in the set of local addresses (e.g., an updated IP address, a network device turned on/off) and realign the content of the Chain Selector with the proper state of the system.

While this simple solution suffices for most common processing case, it would not be able to support the several processing processing paths allowed by the netfilter framework, e.g., when the Linux host is configured to *bridge* the packets between two interfaces (this option is not shown in Figure 2) for the sake of simplicity. As a consequence, the emulation of the filtering behavior of iptables in eBPF may become rather complicated in case a 100% compatibility with iptables is required. In that case, a more effective solution would be to extend the netfilter framework with additional eBPF hook points, which would allow to intercept packets exactly in the desired position of network stack. This would greatly simplify the integration of eBPF-based components with the existing kernel native modules.

Matching algorithm

Iptables uses a linear search algorithm for matching traffic, which is the main responsible for its poor performance particularly in presence of a high number of firewall rules. However, the selection and implementation of a better matching algorithm prove to be a challenging choice due to the intrinsic limitation of the eBPF environment [4]. In fact, although better matching algorithms are well-known in the literature (e.g., cross-producting, decision-tree approaches, etc.), they require either sophisticated data structures that are not currently available in eBPF or an unpredictable amount of memory, which is not desirable for a kernel module.

Given the above constraints, the current prototype of bpf-iptables exploits the Bit Vector Linear Search [3] (LBVS) algorithm, which proves to be reasonably fast while being feasible with current Linux kernels (and available eBPF maps). This algorithm follows the *divide-and-conquer* paradigm: it splits filtering rules into multiple classification steps, based on the number of protocol fields present in the rule set; intermediate results are combined to obtain the final solution.

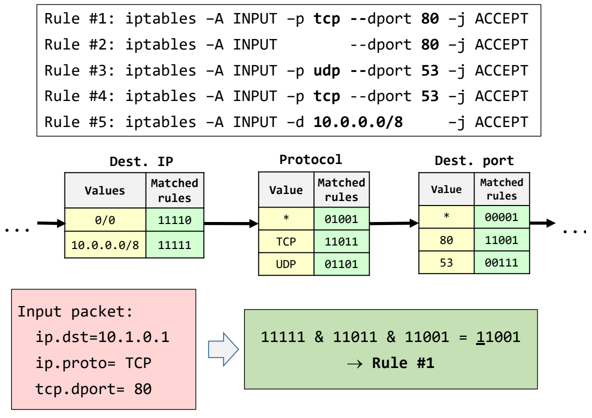
In fact, LBVS creates a specific (logical) bi-dimensional table for each field on which packets may match, such as the three fields (IP destination address, transport protocol, TCP/UDP destination port) shown in the example of Figure 3. Each table contains the list of unique values for that field present in the given ruleset, plus a wildcard for rules that do not care for any specific value. Each value in the table is associated with a bitvector of length N equal to the number of rules, which keeps the list of rules that are satisfied when the field assumes the given value. Filtering rules, and the corresponding bits in the above bitvector, are ordered with highest priority rule first; hence, rule #1 corresponds to the most significant bit in the bitvector. As an example on how the bitvectors are created, rule #5 simply checks the IP destination address and ignores the value of other fields; hence, the 5^{th} bit in each bitvector is `true` when the IP destination address is in range `10.0.0.0/8`, for whatever value of transport protocol and TCP/UDP port (hence, the 5^{th} bit is always 1 for all values in the other two tables).

This matching process is repeated for each field we are operating with, such as the three fields shown in Figure 3. The final matching rule can be obtained by performing a `bitwise AND` operation on all the intermediate bitvectors returned in the previous steps; the resulting rule corresponds to the most significant bit with a value equal to '1' in the resulting bitvector, which represents the matched rule with the highest priority. In the example in Figure 3, it corresponds to the rule #1.

Each processing step is independent, hence 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. Then, the entire set of rules is split into these tables and the classification is carried out in different steps whose results (as bitvectors that maintain the list of matching rules for each field) are combined to obtain the final solution, as shown in Figure 3. In each matching step, if a lookup fails, the algorithm can infer that no match has been found and it can apply the default action, skipping the rest of the pipeline.

The above logical pipeline has been implemented by means of a cascade of eBPF programs as shown in Figure 4, calling each other by means of *tail calls*. A first module is dedicated to the extraction of the packet headers in order to facilitate the processing of the following blocks. Each matching step, which operates on a single field, is translated into a dedicated eBPF program that integrates the required processing code with the most appropriate bi-dimensional per-CPU map, which keeps the couples `value-bitvector` associated to the given field. In each field matching step, the extracted bitvector is compared with the one already obtained at the previous step and stored in a shared per-CPU array, which will be used by the following blocks in the pipeline. Finally, the last block scans the final bitvector looking for the most significant bit at 1; when this is found, it implements the action associated to the rule (drop / accept) and it updates the structure that keeps the counters associated to each rule.

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. For instance, if the TCP flags field is not used by any rule, that processing block is avoided in the pipeline; new processing blocks can be added at run-time if the matching against a new field is required, with the property of running always the optimal number of eBPF programs.



```
Rule #1: iptables -A INPUT -p tcp --dport 80 -j ACCEPT
Rule #2: iptables -A INPUT        --dport 80 -j ACCEPT
Rule #3: iptables -A INPUT -p udp --dport 53 -j ACCEPT
Rule #4: iptables -A INPUT -p tcp --dport 53 -j ACCEPT
Rule #5: iptables -A INPUT -d 10.0.0.0/8     -j ACCEPT
```

Dest. IP

Values	Matched rules
0/0	11110
10.0.0.0/8	11111

Protocol

Value	Matched rules
*	01001
TCP	11011
UDP	01101

Dest. port

Value	Matched rules
*	00001
80	11001
53	00111

```
Input packet:
  ip.dst=10.1.0.1
  ip.proto= TCP
  tcp.dport= 80
```

```
11111 & 11011 & 11001 = 11001
        → Rule #1
```

Figure 3: Linear Bit Vector Search.

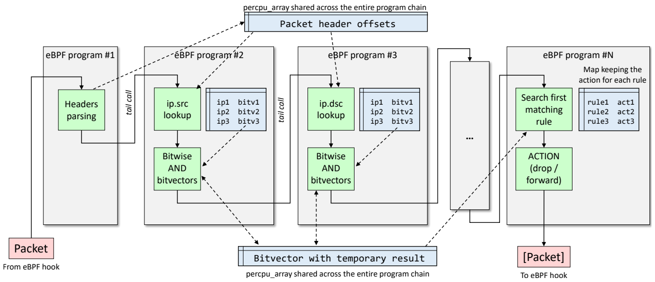


Figure 4: eBPF maching pipeline.

Connection tracking

Netfilter tracks the state of TCP/UDP/ICMP connections and stores them in a session (or connection) table (*conntrack*). This table can be used by iptables to specify filtering rules that accept/drop packets based on the characteristic of the connection they belong to. For instance, iptables may have a rule that allows only packets belonging to a *new* or *established* connections, e.g., enabling the host to generate traffic toward the Internet (and to receive return packets), while connections initiated from the outside world may be forbidden.

In addition of being associated to a connection, each packet can also trigger a state change in a connection; for example, a TCP `SYN` triggers the creation of a new entry in the connection table, while a `RST` packet (which represents the termination of an established connection) flushes an existing entry.

Given the impossibility to exploit the connection tracking facility of the Linux kernel, our bpf-iptables prototype implements its own connection tracking module as a set of eBPF programs. However, due to the well known limitation e.g., in terms of code complexity allowed by this technology, we support basic connection tracking for stateful filtering of UDP, TCP, ICMP traffic that detects when a connection starts/ends, while we do not recognize additional states in the protocol state machines as well as we do not support advanced features such as *related* connections (e.g., when a SIP control session triggers the establishment of voice/video RTP sessions), nor we support IP reassembly. A possible more complete solu-

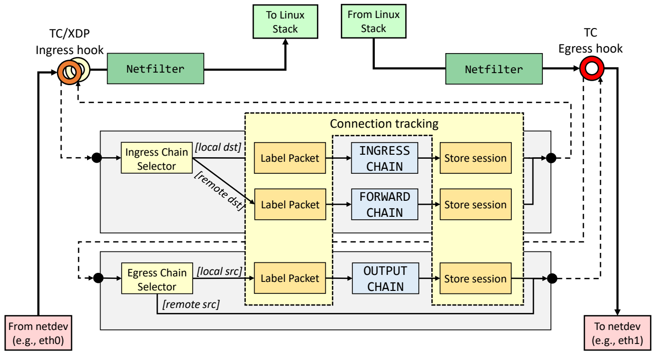


Figure 5: Main filtering architecture of ebpf-iptables (with connection tracking).

tion for connection tracking in eBPF could consist in adding a proper eBPF helper in the kernel that enables to access to conntrack table; this idea has been proposed recently on the netfilter developers mailing list[1].

The resulting architecture for connection tracking in bpf-iptables is shown in Figure 5 and it consists of two eBPF programs. The first program receives the traffic before any chain and (*i*) detects if the incoming packet triggers any change in the conntrack table (e.g., a TCP SYN packet that starts a new connection) and (*ii*) associates the given packet to a session entry, which may be used by the following modules to filter the packet itself based on the above condition. The second program operates at the end of the chain and it stores permanently the state that has been possibly updated by the current packet in the conntrack table. In fact, the new state of the connection (or, in case of a new connection, the associated new entry) is stored in the conntrack table only if the packet survives the filtering; hence, no change occur if the packet is dropped.

The connection tracking module introduces an additional hidden overhead due to the necessity to intercept traffic in both directions (i.e., host to Internet and viceversa), which is needed in order to follow the protocol state machine, e.g., to recognize the three way handshake (SYN, SYN/ACK, ACK) sequence in order to declare a new TCP connection as "established". In other words, even if the user installs a set of iptables rules that operate only on the INPUT chain, a minimal eBPF processing pipeline that encompasses only the connection tracking modules is enabled anyway on the TC_EGRESS hook.

An additional problem found in the implementation of the bpf-iptables conntrack module was the difficulty in cleaning up the connection table, e.g., to purge zombie sessions. In fact, since eBPF programs are event-driven, the natural place to put such a function would be the control plane, with a thread dedicated only to clean expired entries. However, this solution may lead to race conditions, as there is no locking mechanism that allow to define a critical section shared between kernel eBPF programs and userspace programs; this results in the possibility that an entry is deleted by the userspace while the eBPF program is actually making use of the content of that entry.

The current implementation circumvent this problem by implementing the conntrack table with an *LRU map*, so that old entries are automatically recycled and assigned to new connections. In addition, it stores in each entry a timestamp representing the last time that entry was used; this enables to detect a possible access to an old entry, not yet purged by the LRU algorithm. This is not the optimal solution, but it represents the best compromise given the possibilities offered by the current eBPF technology.

Preserving iptables syntax

The compatibility with the iptables syntax is another must for this prototype. This enables potential users to exploit the same tools/scripts they currently use for controlling iptables to interact also with bpf-iptables, providing a smooth migration experience. However, since a full clone of iptables may not be feasible in the short term, we would like to guarantee users they will always obtain exactly the result they expect, even if the current bpf-iptables may not be able to support all issued commands.

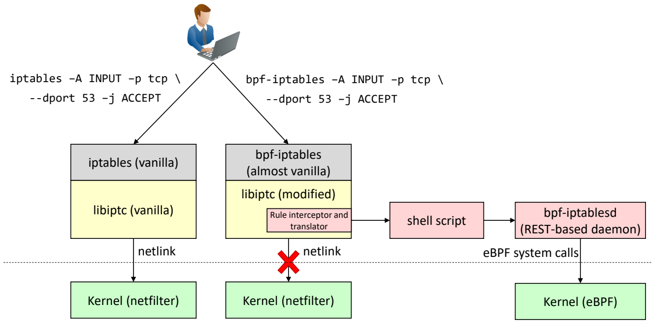
Our solution to the above problems was to create two executables, `iptables` and `bpf-iptables`, which are available at the same time in the system, the former controlling the traditional filtering based on netfilter, the second emulating iptables by means of the eBPF clone. Ideally, both tools should support the same syntax; in practice the latter supports only a subset of commands compared to the former due to the scarce maturity of our solution. This allow users to either call `iptables` or `bpf-iptables`, using the same syntax; in case a command is not supported by the latter, the user can always switch back to the original iptables and obtain the network behavior he needs. This solution is very simple and leaves the responsibility to choose the right executable to the user; while more sophisticated solution can be envisioned, this may be acceptable in the short term.

Technically, this has been achieved by a lightweight set of modifications to the iptables source code, which has been cloned and renamed as bpf-iptables, and to the underlying `libiptc` library. Our modifications simply change the way iptables and `libiptc` push commands in the kernel, replacing `netlink` messages with an equivalent command line that is passed to an intermediate shell script in charge of calling the `bpf-iptablesd` daemon, which implements the actual eBPF-based iptables clone. This daemon has a REST interface waiting for JSON commands that ask to add/remove filtering rules, read the state of the system (e.g., statistics/counters) and more; the above commands are translated in eBPF-compatible primitives that are sent to the kernel, e.g., to generate a new set of eBPF programs[2], to read data in a map, and more.

This approach has the advantage of introducing no additional cost cost for parameter parsing and validation, already

[1] `https://www.mail-archive.com/netfilter-devel@vger.kernel.org/msg11139.html`

[2] Due to the LBVS internals and the way we create eBPF programs, we have to generate a new set of eBPF programs each time there is a change in the filtering rule dataset.



```
iptables -A INPUT -p tcp \
  --dport 53 -j ACCEPT
```

```
bpf-iptables -A INPUT -p tcp \
  --dport 53 -j ACCEPT
```

Figure 6: Userspace integration of bpf-iptables.

performed by iptables, as well as limiting the number of bugs, as we reuse already existing and well tested code for the frontend. An overview of the overall architecture is depicted in Figure 6.

Conclusions

This paper presents a preliminary architecture for a possible replacement of iptables with an equivalent software based on the eBPF technology.

While proven to be feasible, the prototype highlighted also the complexity of creating a full clone of iptables, in particular considering that this paper addressed only a subset of the features available in that software. For instance, iptables is often used to handle both *filtering* and *natting* functions, with the latter not been considered in this paper, as well as the features available in ebtables and arptables, and the additional packet paths in netfilter when bridging (instead of IP forwarding) is enabled.

Starting from the above observation, we can envision two possible direction for our future work.

First, instead of trying to substitute iptables with a full eBPF clone, it would be worth exploring the possibility of offloading a subset of the filtering rules in an eBPF program, running at high speed (e.g., in XDP). This may be the case for long lists of homogeneous rules (e.g., operating on IP destination address) that are used to discard traffic from malicious sources; instead of matching each incoming packet against that set of rules, which are processed with the linear search available in iptables, those could be moved to a processing module that implements a more efficient algorithm and runs earlier in the network stack, as long as the semantic of the rules is preserved. This results in having malicious packets dropped earlier, with the consequent reduced resource (CPU) consumption.

Second, having a more extensive set of hook points operating in netfilter could add more flexibility in integrating eBPF programs in Linux, as it would enable the selective replacement of a single component while keeping the others unchanged (e.g., replace only the firewall, but keep the Linux IP forwarding). This flexibility is not available with the current eBPF hooks; for instance, if the filtering is implemented with eBPF, the NAT has to be re-implemented in eBPF as well in order to preserve the semantic of the rules. In fact, if we take the `INPUT` chain as example and assuming that the filtering is done in eBPF, the current NAT would operate on packets exiting from the filtering components, while in the original iptables the NAT is traversed before packets arrive to the filtering block.

Acknowledgments

The authors would like to thank the other people who contributed to this work, particularly Mauricio Vásquez Bernal who developed part of the software framework that was used to implement this prototype.

This was work was possible thanks to the generous support from Huawei Technologies and Vmware. Part of this work was conducted within the framework of the ASTRID project, which has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 786922. Study sponsors had no role in writing this paper. The views expressed do not necessarily represent the views of the authors employers, the ASTRID project, or the Commission of the European Union.

References

[1] Borkmann, D. 2018. net: add bpfilter. [Online; last-retrieved 30-June-2018].

[2] Corbet, J. 2009. Nftables: a new packet filtering engine. [Online; last-retrieved 30-June-2018].

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

[4] Miano, S.; Bertrone, M.; Risso, F.; Vásquez Bernal, M.; and Tumolo, M. 2018. Creating complex network service with ebpf: Experience and lessons learned. In *High Performance Switching and Routing (HPSR)*. IEEE.

[5] Russell, P. 1998. The netfilter.org project. [Online; last-retrieved 30-June-2018].

[6] Wallen, J. 2015. An introduction to uncomplicated firewall (ufw). [Online; last-retrieved 30-June-2018].