Securing Linux with a Faster and Scalable Iptables

Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal
Dept. of Computer and Control Engineering,
Politecnico di Torino, Italy
name.surname@polito.it

Yunsong Lu, Jianwen Pi
Huawei Technologies, Inc., Santa Clara, CA
roamer@yunsong.lu,jianwpi@gmail.com

Under submission

ABSTRACT

The sheer increase in network speed and the massive deployment of containerized applications in a Linux server has led to the consciousness that `iptables`, the current de-facto firewall in Linux, may not be able to cope with the current requirements particularly in terms of scalability in the number of rules. This paper presents an eBPF-based firewall, `bpf-iptables`, which emulates the `iptables` filtering semantic while guaranteeing higher throughput. We compare our implementation against the current version of `iptables` and other Linux firewalls, showing how it achieves a notable boost in terms of performance particularly when a high number of rules is involved. This result is achieved without requiring custom kernels or additional software frameworks (e.g., DPDK) that could not be allowed in some scenarios such as public data-centers.

CCS CONCEPTS

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

KEYWORDS

eBPF, iptables, Linux, XDP

1 INTRODUCTION

Nowadays, the traditional security features of a Linux host are centered on `iptables`, which allows applying different security policies to the traffic, such as to protect from possible network threats or to prevent specific communication patterns between different machines. Starting from its introduction in kernel v2.4.0, `iptables` remained the most used packet filtering mechanism in Linux, despite being strongly criticized under many aspects, such as for its far from cutting-edge matching algorithm (i.e., linear search) that limits its scalability in terms of number of policy rules, its syntax, not always intuitive, and its old code base, which is difficult to understand and maintain. In the recent years, the increasing demanding of 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 performance, flexibility, and scalability [18].

`Nftables` [11] was proposed in 2014 with the aim of replacing `iptables`; it reuses the existing `netfilter` subsystem through an in-kernel virtual machine dedicated to firewall rules, which represents a significant departure from the `iptables` filtering model. In particular, it (*i*) integrates all the functionalities provided by `{ip, ip6, arp, eb}tables`; (*ii*) uses a nicer syntax; (*iii*) improves the classification pipeline introducing maps and concatenations, allowing to construct the ruleset in a way that reduces the number of memory lookups per packet before finding the final action to apply; finally, (*iv*) it moves part of the intelligence in the userspace `nft` tool, which is definitely more complex than `iptables` but allows to potentially deliver new features or protocols without kernel upgrades [4]. Although this yields advantages over its predecessor, `nftables` (and other previous attempts such as `ufw` [38] or `nf-HiPAC` [29]) did not have the desired success, mainly due to the reluctance of the system administrators to adapt their existing configurations (and scripts) operating on the old framework and move on to the new one [12]. This is also highlighted by the fact that the majority of today's open-source orchestrators (e.g., Kubernetes [22], Docker [21]) are strongly based on `iptables`.

Recently, another in-kernel virtual machine has been proposed, the extended BPF (eBPF) [2, 17, 35], which offers the possibility to dynamically generate, inject and execute arbitrary code inside the Linux kernel, without the necessity to install any additional kernel module. eBPF programs can be attached to different hook points in the networking stack such as eXpress DataPath (XDP) [20] or Traffic Control (TC), hence enabling arbitrary processing on the intercepted packets, which can be either dropped or returned (possibly modified) to the stack. Thanks to its flexibility and excellent performance, functionality, and security, recent activities on the Linux networking community have tried to bring the power of eBPF into the newer `nftables` subsystem [6]. Although this would enable `nftables` to converge towards an implementation of its VM entirely based on eBPF, the proposed design does not fully exploit the potential of eBPF, since the programs are directly generated in the kernel and not in userspace, thus losing all the separation and security properties guaranteed by the eBPF code verifier that is executed before the code is injected in the kernel.

On the other hand, `bpfilter` [8] proposes a framework that enables the transparent translation of existing `iptables` rules into eBPF programs; system administrators can continue to use the existing `iptables`-based configuration without even knowing that the filtering is performed with eBPF. To enable such design, `bpfilter` introduces a new type of kernel module that delegates its functionality into user space processes, called *user mode helper* (`umh`), which can implement the rule translation in userspace and then inject the newly created eBPF programs in the kernel. Currently, this work focuses mainly on the design of a translation architecture for `iptables` rules into eBPF instructions, with a small proof of concept that showed the advantages of intercepting (and therefore

filtering) the traffic as soon as possible in the kernel, and even in the hardware (smartNICs) [37].

The work presented in this paper continues along the `bpfilter` proposal of creating a faster and more scalable clone of `iptables`, but with the following two additional challenges. First is to **preserve the iptables filtering semantic**. Providing a transparent replacement of `iptables`, without users noticing any difference, imposes not only the necessity to respect its syntax but also to implement exactly its behavior; small or subtle differences could create serious security problems for those who use `iptables` to protect their systems. Second is to **improve speed and scalability** of `iptables`; in fact, the linear search algorithm used for matching traffic is the main responsible for its limited scalability particularly in the presence of a large number of firewall rules, which is perceived as a considerable limitation from both the latency and performance perspective.

Starting from the above considerations, this paper presents the design of an eBPF-based clone of `iptables`, called **`bpf-iptables`**, which implements an alternative filtering architecture in eBPF, while maintaining the same `iptables` filtering semantic and with improved performance and scalability. Being based on the eBPF subsystem, `bpf-iptables` can leverage any possible speedup available in the Linux kernel to improve the packet processing throughput. Mainly, XDP is used to provide a fast path for packets that do not need additional processing by the Linux stack (e.g., packets routed by the host) or to discard traffic as soon as it comes to the host. This avoids useless networking stack processing for packets that must be dropped by moving *some* firewall processing off the host CPU entirely, thanks to the work that has been done to enable the offloading at XDP-level [7].

Our contributions are: *(i)* the design of `bpf-iptables`; it provides an overview of the main challenges and possible solutions in order to preserve the `iptables` filtering semantic given the difference, from hook point perspective, between eBPF and `Netfilter`. To the best of our knowledge, `bpf-iptables` is the first application that provides an implementation of the `iptables` filtering in eBPF. *(ii)* A comprehensive analysis of the main limitations and challenges required to implement a fast matching algorithm in eBPF, keeping into account the current limitations [28] of the above technology. *(iii)* A set of data plane optimizations that are possible thanks to the flexibility and dynamic compilation (and injection) features of eBPF, allowing us to create at runtime an optimized data path that fits perfectly with the current ruleset being used.

In the rest of this paper, we present the challenges, design choices and implementation of `bpf-iptables` and we compare it with the current implementations of `iptables` and `nftables`. For this paper, we take into account only the support for the `FILTER` table, while we leave as future work the support for additional features such as `NAT` or `MANGLE`.

2 DESIGN CHALLENGES

This Section introduces the main challenges we encountered while designing `bpf-iptables`, mainly derived from the necessity to emulate the `iptables` implementation with the eBPF technology.

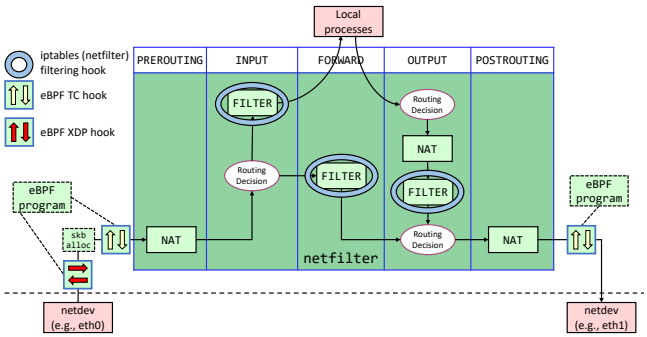


Figure 1: Location of `netfilter` and eBPF hooks.

2.1 Guaranteeing filtering semantic

The main difference between `iptables` and `bpf-iptables` lies in their underlying frameworks, `netfilter` and eBPF respectively. `Iptables` defines three default chains for filtering rules associated to the three `netfilter` hooks [31] shown in Figure 1, which allow to filter traffic in three different locations of the Linux networking stack. Particularly, those hook points filter traffic that (*i*) *terminates* on the host itself (`INPUT` chain), (*ii*) *traverses* the host such as when it acts as a router and forwards IP traffic between multiple interfaces (the `FORWARD` chain), and (*iii*) *leaves* the host (`OUTPUT` chain).

On the other hand, eBPF programs can be attached to different hook points. As shown in Figure 1, ingress traffic is intercepted in the XDP or traffic control (`TC`) module, hence *earlier* than `netfilter`; the opposite happens for outgoing traffic, which is intercepted *later* than `netfilter`. The different location of the filtering hooks in the two subsystems introduces the challenge of preserving the semantic of the rules, which, when enforced in an eBPF program, operate on a different set of packets compared to the one that would cross the same `netfilter` chain. For example, rule "`iptables -A INPUT -j DROP`" drops all the incoming traffic crossing the `INPUT` chain, hence directed to the current host; however, it does not affect the traffic forwarded by the host itself, which traverses the `FORWARD` chain. A similar "drop all" rule, applied in the `XDP` or `TC` hook, will instead drop *all* the incoming traffic, including packets that are forwarded by the host itself. As a consequence, `bpf-iptables` must include the capability to predict the `iptables` chain that would be traversed by each packet, maintaining the same semantic although attached to a different hook point.

2.2 Efficient classification algorithm in eBPF

The selection and implementation of a better matching algorithm proved to be challenging due to the intrinsic limitations of the eBPF environment [28]. In fact, albeit better matching algorithms are well-known in the literature (e.g., cross-producting [34], decision-tree approaches [15, 19, 30, 32, 33, 36]), they require either sophisticated data structures that are not currently available in eBPF[1] or an unpredictable amount of memory, which is not desirable for a module

[1] eBPF programs do not have the right to use traditional memory; instead, they need to rely on a limited set of predefined memory structures (e.g., hash tables, arrays, and a few others), which are used by the kernel to guarantee safety properties and possibly avoid race conditions. As a consequence, algorithms that require different data structures are not feasible in eBPF.

operating at the kernel level. Therefore, the selected matching algorithm must be efficient and scalable, but also feasible with the current eBPF technology.

2.3 Support for stateful filters (conntrack)

`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 stateful rules that accept/drop packets based on the characteristic of the connection they belong to. For instance, `iptables` may have a rule that accepts only outgoing packets belonging to `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. As shown in Figure 1, `bpf-iptables` operates *before* packets enter in `netfilter` and therefore it cannot exploit the Linux *conntrack* module to track incoming and outgoing packets. As a consequence, it has to implement its own connection tracking module, which enables filtering rules to operate also on the state of the connection, as described in Section 4.5.

3 OVERALL ARCHITECTURE

Figure 2 shows the overall system architecture of `bpf-iptables`. The data plane includes three main categories of eBPF programs. The first set, shown in blue, implements the classification pipeline (i.e., the ingress, forward or output chain); a second set, in yellow, implements the logic required to guarantee the semantics of `iptables`; a third set, in orange, is dedicated to connection tracking. Additional programs, in grey, are dedicated to ancillary tasks such as packet parsing.

The *ingress* pipeline is called upon receiving a packet either on the XDP or TC hook. By default, `bpf-iptables` attaches eBPF programs to the XDP hook, which triggers the execution of the filtering pipeline immediately after the arrival of a packet on the NIC. However, this requires the explicit support for XDP in the NIC driver[2]; `bpf-iptables` automatically falls back to the TC hook when the NIC driver is not XDP-compatible. If the Linux host has multiple NICs to attach to, `bpf-iptables` can selectively attach to either XDP/TC hooks, hence guaranteeing fast processing on XDP-compatible NICs while providing transparent support for other NICs as well. The *egress* pipeline is instead called upon receiving a packet on the TC egress hook, before the packet leaves the host; XDP is not available in this transmission path[3].

Once in the *ingress* pipeline, the packet can enter either the `INPUT` or `FORWARD` chain depending on the routing decision; in the first case, if the packet is not dropped, it will continue its journey through the Linux TCP/IP stack, ending up in a local application. In the second case, if the `FORWARD` pipeline ends with an `ACCEPT` decision, `bpf-iptables` redirects the packet to the target NIC, without returning to the Linux networking stack (more details will be presented in Section 4.4.2). On the other hand, a packet leaving the host triggers the execution of `bpf-iptables` when it reaches the TC egress hook. In such event, `bpf-iptables` has to find whether the outgoing packet comes from a local application and therefore must be processed by the `OUTPUT` chain, or the host has forwarded it; in the latter case, it is passed as it is, without further processing.

Finally, a control plane module (not depicted in Figure 2) is executed in userspace and provides three main functions: *(i)* initialization and update of the `bpf-iptables` data plane, *(ii)* configuration of the eBPF data structures required to run the classification algorithm and *(iii)* monitoring for changes in the number and state of available NICs, which is required to fully emulate the behavior of `iptables`, handling the traffic coming from *all* the host interfaces. We will describe the design and architecture of the `bpf-iptables` data plane in Section 4, while the operations performed by the control plane will be presented in Section 5.

4 DATA PLANE

In the following subsections we present the different components belonging to the `bpf-iptables` dataplane, as shown in Figure 2.

4.1 Header parsing

The `bpf-iptables` ingress and egress pipeline start with a *Header Parser* module that extracts the packet headers required by the current filtering rules, and stores each field value in a per-CPU array map shared among all the eBPF programs in the pipeline, called *packet metadata*. This avoids the necessity of packet parsing capabilities in the subsequent eBPF programs and guarantees both better performance and a more compact processing code. The code of the *Header Parser* is dynamically generated on the fly; when a new filtering rule that requires the parsing of an additional protocol field is added, the control plane re-generates, compiles and re-injects the obtained eBPF program in the kernel in order to extract also the required field. As a consequence, the processing cost of this block is limited exactly to the number of fields that are currently needed by the current `bpf-iptables` rules.

4.2 Chain Selector

The *Chain Selector* is the second module encountered in the `bpf-iptables` pipeline. It has the responsibility to classify and forward the traffic to the correct classification pipeline (i.e., chain), in order to preserve the `iptables` semantic, answering to the problem presented in Section 2.1. It anticipates the routing decision that would have been performed later in the TCP/IP stack and is, therefore, able to predict the right chain that will be hit by the current packet. The idea is that traffic coming from a network interface would cross the `INPUT` chain only if it is directed to a *local* IP address, visible from the host root namespace, while incoming packets directed to a *remote* IP address would cross the `FORWARD` chain. On the other hand, an outgoing packet would traverse the `OUTPUT` chain only if it has been generated locally, i.e., by a *local* IP address.

To achieve this behavior, `bpf-iptables` uses a separate Chain Selector module for the ingress and egress pipeline. The Ingress Chain Selector classifies traffic based on the *destination* IP address of the traversed packet; it checks if the address is present in the `BPF_HASH` map that keeps local IPs and then sends the traffic to either the `INPUT` or `FORWARD` chain. These operations are delegated to two different instances of the Ingress Chain Selector, which represent two distinct eBPF programs. The first program (Part I) performs the actual classification of the packet and decides the target chain,

[2]NIC driver that have native support for XDP can be found at [9].
[3]A proposal for adding an XDP Egress hook in the future was put forward and is under discussion in the Linux kernel community [10].

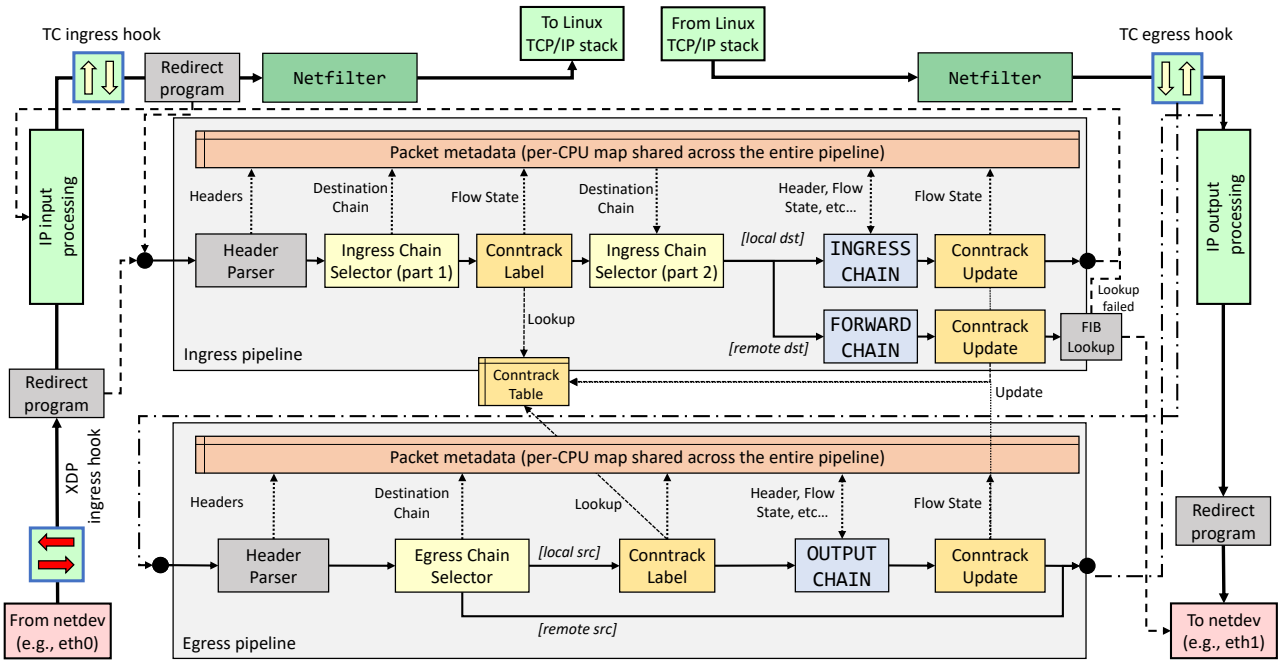


Figure 2: High-level architecture of `bpf-iptables`.

which is written in the *packet metadata* per-CPU map shared across the entire pipeline. When triggered, the second program (Part II), reads the destination chain from the per-CPU map and forwards the packet to the correct chain, by performing a tail call to the first program of the target classification pipeline. Although this process could be performed within a single eBPF program, we decided to split the two tasks in order to allow the subsequent program of the pipeline (i.e., the connection tracking module) to benefit from the knowledge of the target chain and adopt several optimizations that would improve the `bpf-iptables` processing speed. For example, if the first rule of the `INPUT` chain requires to *accept* all the `ESTABLISHED` connections, the connection tracking module, after reading the target chain (e.g., `INPUT`), can directly jump at the end of the classification pipeline, without further processing.

On the other hand, the Egress Chain Selector, which is part of the egress pipeline, classifies traffic based on the *source* IP address and sends it to either the `OUTPUT` chain, or directly to the output interface[4].

4.3 Matching algorithm

To overcome the performance penalties of the linear search adopted by `iptables`, `bpf-iptables` adopts the more efficient Linear Bit-Vector Search (LBVS) [25]) classification algorithm. LBVS provides a reasonable compromise between feasibility and speed; it has an intrinsic pipelined structure which maps nicely with the eBPF technology, hence making possible the optimizations presented in Section 4.4.2. The 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 ruleset; intermediate results that carry the potentially matching rules are combined to obtain the final solution.

Classification. LBVS requires 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, in which the i^{th} '1' bit tells that rule i can be possibly matched when the field assumes that 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.

The 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 and determining the most significant '1' bit in the resulting bitvector. This represents the matched rule with the highest priority, which corresponds to rule #1 in the example in Figure 3. Bitmaps enable the evaluation of rules in large batches, which depend on the parallelism of the main memory; while still theoretically a linear algorithm, this scaling factor enables a 64x speedup compared to a traditional linear search on common CPUs.

[4]Traffic traversing the `FORWARD` chain has already been matched in the ingress pipeline, hence it should not be matched in the `OUTPUT` chain.

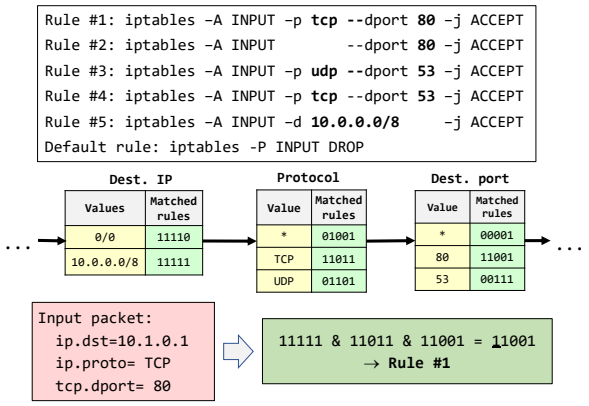


Figure 3: Linear Bit Vector Search

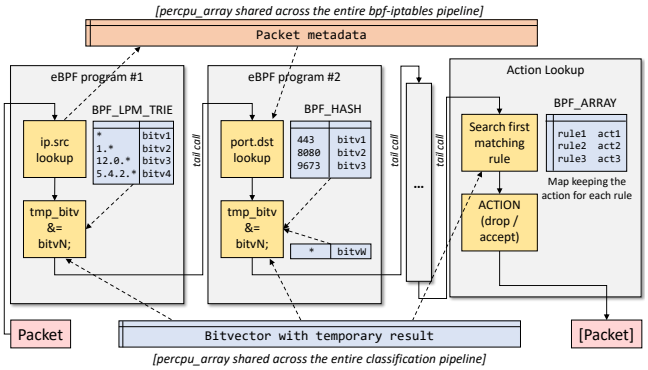


Figure 4: `bpf-iptables` classification pipeline.

4.4 Classification Pipeline

A packet leaving the Chain Selector enters in the target classification pipeline, shown in Figure 4, whose role is to filter the packet according to the rules configured for that chain.

The `bpf-iptables` classification pipeline is made by a cascade of eBPF programs calling each other by means of *tail calls* where every single program manages a single matching protocol field of the current ruleset. The pipeline contains a per-CPU map shared among all the programs belonging to the same chain that is used to share some information between the programs of the chain, such as the temporary bitvector containing the partial matching result, which is initialized with all the bits set to 1 before a packet enters the pipeline.

Every single module of the pipeline performs the following operations: *(i)* extracts the needed packet fields from the *packet metadata* per-CPU map, previously filled by the *Header Parser* module; *(ii)* performs a lookup on its own eBPF map to find the bitvector associated to the current packet value for that field and *(iii-a)* if the lookup succeeds, performs a `bitwise AND` between this bitvector and the *temporary* bitvector contained in the per-CPU map. If the lookup fails and there is a wildcard rule, *(iii-b)* the `AND` is performed between the bitvector associated with the wildcard rules and the one present in the per-CPU map. Instead, *(iii-c)* if the lookup fails and there are not wildcard rules for that field, we can directly conclude that the current packet does not match any rule within the ruleset; in this event, the default action for that chain is applied, skipping all the subsequent modules of the pipeline. Finally, except the last case, *(iv)* it performs a *tail-call* to the next module of the chain, after saving the bitvector into the per-CPU map.

Bitvectors comparison. Since every matching rule is represented as a 1 in the bitvector, `bpf-iptables` uses an array of N 64bit unsigned integers to support a large number of rules (e.g., 2.000 rules can be represented as an array of 32 `uint64_t`). As consequence, when performing the `bitwise AND`, the current eBPF program has to perform N cycles on the entire array to compare the two bitvectors. Given the lack of loops on eBPF, this process requires loop unrolling and is therefore limited by the maximum number of possible instructions within an eBPF program, thus also limiting the overall number of rules supported by `bpf-iptables`[5]. The necessity of perform loop unrolling is, as consequence, the most compelling reason for splitting the classification pipeline of `bpf-iptables` across many eBPF modules, instead of concentrating all the processing logic within the same eBPF program.

Action lookup. Once we reach the end of the pipeline, the last program has to find the rule that matched the current packet. This program extracts the bitvector from the per-CPU shared map and looks for the position of the first bit to 1 in the bitvector, using the de Bruijn sequences [26] to find the index of the first bit set in a single word; once obtained, it uses that position to retrieve the final action associated with that rule from a given `BPF_ARRAY` map and finally applies the action. Obviously, if no rules have been matched, the default action is applied.

4.4.1 Clever data sharing. `bpf-iptables` makes a massive use of eBPF per-CPU maps, which represent memory that can be shared among different cascading programs, but that exist in multiple independent instances equal to the number of available CPU cores. This memory structure guarantees very fast access to data, as it statically assigns a set of memory locations to each CPU core; consequently, data is never realigned with other L1 caches present on other CPU cores, hence avoiding the (hidden) hardware cost of cache synchronization. Per-CPU maps represent the perfect choice in our scenario, in which multiple packets can be processed in parallel on different CPU cores, but where all the eBPF programs that are part of the same chain are guaranteed to be executed on the same CPU core. As a consequence, all the programs processing a packet P are guaranteed to have access to the same shared data, without performance penalties due to possible cache pollution, while multiple processing pipelines, each one operating on a different packet, can be executed in parallel. The consistency of data in the shared map is guaranteed by the fact that eBPF programs are never preempted by the kernel (even across tail calls). They can use the per-CPU map as a sort of stack for temporary data, which can be subsequently obtained from the downstream program in the chain with the guarantees that data are not overwritten during the parallel execution of another eBPF

[5]The possibility to perform *bounded* loops in the eBPF program, which is work in progress under the Linux kernel community, would reduce this limit [13, 14].

program on another CPU and thus ensuring the correctness of the processing pipeline.

4.4.2 Pipeline optimizations. Thanks to the modular structure of the bpf-iptables pipeline and the possibility to re-generate and compile part of it at runtime, we can adopt several optimizations that allow *(i)* to jump out of the pipeline when we realize that the current packet does not require further processing and *(ii)* to modify and rearrange the pipeline at runtime based on the current bpf-iptables ruleset values.

Early-break. While processing a packet into the classification pipeline, bpf-iptables can discover in advance that it will not match any rule. This can happen in two separate cases. The first occurs when, at any step of the pipeline a lookup in the bitvector map fails; in such event, if that field does not have a wildcard bitvector, we can directly conclude that the current packet will not match any rule.

The second case takes place when the result of the bitwise *AND* between the two bitvectors is an empty bitvector (all bits set to 0). In both circumstance, every single module can detect this situation and jump out of the pipeline by applying the default policy for the chain, without the additional overhead of executing all the other component, since no rules matched the current packet. If the policy is DROP, the packet is immediately discarded concluding the pipeline processing; if the default policy is ACCEPT, the packet will be delivered to the destination, before being processed by the *Conntrack Update* module (Section 4.5).

Accept all established connections. A common configuration applied in most iptables rulesets contains an ACCEPT all ESTABLISHED connections as the first rule of the ruleset. When the bpf-iptables control plane discovers this configuration in a chain, it forces the *Conntrack Label* program to skip the classification pipeline if it recognizes that a packet belongs to an ESTABLISHED connection. Since this optimization is performed per-chain (we could have different configurations among the chains), the Conntrack Label module reads the target chain from the *packet metadata* per-CPU map previously filled by the Chain Selector and directly performs a *tail-call* to the final connection tracking module that will update the conntrack table accordingly (e.g., updating the timestamp for that connection).

Optimized pipeline. Every time the current ruleset is modified, bpf-iptables creates a processing pipeline that contains the minimum (optimal) number of processing blocks required to handle the fields of the current ruleset, avoiding unnecessary processing. For instance, if there are no rules checking the value of TCP flags, that processing block is not added to the pipeline; new processing blocks can be dynamically added at run-time if the matching against a new field is required. In addition, bpf-iptables is able to re-organize the classification pipeline by changing the order of execution of the various components that characterize it. For example, if some components require only an exact matching, a match failed on that field would lead to an early-break of the pipeline, as presented before; putting those modules at the beginning of the pipeline could speed up processing, avoiding unnecessary memory accesses and additional processing.

HOmogeneous RUleset analySis (HORUS). The HORUS optimization is used to (partially) overcome two main restrictions of bpf-iptables: the maximum number of matching rules, given by the necessity to perform loop unrolling to compare the bitvectors, and the rule update time since it is necessary to re-compute the bitvector used in the classification pipeline when the firewall configuration changes. The idea behind **HORUS** is based on the consideration that often, firewall rulesets (in particular, the ones automatically configured by orchestrations software), contain a set of *homogeneous* rules that operate on the same set of fields. If we are able to discover this set of "similar" rules that are not conflicting with the previous ones (with higher priority) we could bring them in front of the matching pipeline to speed up their matching. In addition, since those rules are independent from the others in the ruleset, we could remove their corresponding bits from the bitvectors used in the matching pipeline, increasing the space for other non-HORUS rules. Moreover, if the bpf-iptables control plane discovers that a newly installed (or removed) rule belongs to the HORUS ruleset, it does not need to update or even change the entire matching pipeline, but a single map insertion (or deletion) would be enough, thus reducing the rule update time in a way that is completely independent from the number of rules installed in that chain. When enabled, the HORUS ruleset is inserted right before the Conntrack Label and consists of another eBPF program with a BPF_HASH table that contains as key, the set of fields of the HORUS set and, as value, the final action to apply when a match is found. If the final action is DROP, the packet is immediately dropped, otherwise if the action is ACCEPT it will directly jump to the last module of the pipeline, the *Conntrack Update*, before continuing towards its destination. On the other hand, if a match in the table is not found, HORUS will directly jump to the first program of the classification pipeline, following the usual path.

An important scenario where HORUS shows its great advantages is under **DoS attacks**. In fact, if bpf-iptables detects a HORUS ruleset that contains all the rules with a DROP action, packets will be immediately discarded as soon as the match is found in the HORUS program, hence exploiting *(i)* the early processing provided by XDP that allows to drop packets at a high rate and *(ii)* the ability to run this program on hardware accelerators (e.g., smartNIC) that support the offloading of "simple" eBPF programs, further reducing the system load and the resource consumption.

Optimized forwarding. A packet traversing the FORWARD chain is filtered according to the installed rules. If the final decision for the packet is ACCEPT, it has to be forwarded to the next-hop, according to the routing table of the host. Since eBPF program are running in the kernel, they can directly lookup the kernel data structures in order to find the needed information. In particular, starting from kernel version 4.18, eBPF programs can query the routing table of the host to find the next-hop information; bpf-iptables uses this data to optimize the path of the packet in the FORWARD chain by directly forwarding the packet to the target NIC, shortening its route within the Linux host. This brigs a significant improvement in terms of performance as shown in Section 6. Of course, there are cases where the needed information are not available (e.g., because the MAC address of the next hop is not yet known); in such events, bpf-iptables will deliver the first few packets to the Linux stack, following the usual path.

4.4.3 Atomic rule update. One of the drawbacks of the LBVS classification algorithm is that, whenever a new rule is added, updated or removed from the current ruleset, it triggers the re-computation of the bitvectors associated with the current fields. In the `bpf-iptables` classification pipeline, each matching field is managed by a separate eBPF program, with each program using a separate map. To avoid inconsistency problems, we must update *atomically* the content of *all* maps; if a packet enters the pipeline where only a subset of the maps has been updated, the result of the matching could be unpredictable. Unfortunately, eBPF allows the atomic update of a *single* map, while it does not support atomic updates of multiple maps at the same time. Furthermore, implementing a synchronization mechanism for the update (e.g., using locks to prevent traffic being filtered by `bpf-iptables`) could lead to unacceptable service disruption given the impossibility of the data plane to process the traffic in that time interval.

To solve this issue, `bpf-iptables` duplicates the current classification pipeline as a new chain of eBPF programs and maps reflecting the new configuration; this is possible because the `bpf-iptables` classification pipeline is stateless, hence creating a parallel chain will not require to migrate the old state. While this new pipeline is assembled and injected into the kernel, packets continue to be processed in the old matching pipeline, accessing the old state and configuration; when this reloading phase is completed, the Chain Selector is updated to point to the first program of the new chain, allowing new packets to flow through it. This operation is guaranteed to be *atomic* by the eBPF subsystem, which uses a particular map (`BPF_PROG_ARRAY`) to keep the addresses of the instantiated eBPF programs. Updating the address of the old chain with the new one is performed *atomically*, enabling the continuous processing of the traffic, always with a consistent state and without any service disruption.

Finally, when the new chain is up and running, the new configuration is unloaded. We discuss and evaluate the performance of the rule updates within `bpf-iptables`, `iptables` and `nftables` in Section 6.4.2.

4.5 Connection Tracking

To support stateful filters, `bpf-iptables` implements its own connection tracking module, which is characterized by four additional eBPF programs placed in both ingress and egress pipeline, plus an additional matching component in the classification pipeline that filters traffic based on the current connection's state. These module share the same `BPF_HASH` *conntrack* map, as shown in Figure 2.

To properly update the state of a connection, the `bpf-iptables` connection tracking component has to intercept the traffic in both directions (i.e., host to the Internet and vice versa). Even if the user installs a set of rules operating only on the `INPUT` chain, the packet has to be processed by the connection tracking modules located on the egress pipeline, which are the same as the ones situated into the ingress pipeline but loaded into the `TC_EGRESS` hook. The `bpf-iptables` connection tracking has built-in support for TCP, UDP, and ICMP traffic, although does not handle advanced features such as *related* connections (e.g., when a control FTP connection is used to start a new data FTP connection, or a SIP control session triggers the establishment of voice/video RTP sessions[6]), nor it supports IP reassembly.

Packet walkthrough. The *Conntrack Label* module is used to associate a label to the current packet[7] by detecting any possible change in the *conntrack* table (e.g., `TCP SYN` packet starting a new connection triggers the creation of a new session entry), which is written into the *packet metadata* per-CPU map shared within the entire pipeline. This information is used inside the classification pipeline to filter the packet according to the stateful rules of the ruleset. Finally, if the packet "survives" the classification pipeline, it is sent to the second connection tracking program, called *Conntrack Update*, which updates the *conntrack* tables with the new connection state (or, in the case of a new connection, it creates the new associated entry); consequently, no changes occur if the packet is dropped with the result that forbidden sessions will never consume space in the connection tracking table.

Conntrack entry creation. To identify the connection associated to a packet, `bpf-iptables` uses the packet 5-tuple (i.e., src-dest IP address, L4 protocol, and src-dest L4 port) as key in the *conntrack* table. Before saving the entry in the table, the *Conntrack Update* orders the key according to the formula:

$$key = \{min(IpSrc, IpDest), max(IpSrc, IpDest), Proto, \\ min(PortSrc.PortDest), max(PortSrc, PortDest)\} \quad (1)$$

This process allows to create a single entry in the *conntrack* table for both directions, speeding up the lookup process. In addition, together with the new connection state, the *Conntrack Update* module stores into the *conntrack* table two additional flags, *ip reverse* (`ipRev`) and *port reverse* (`portRev`) indicating if the IPs and the L4 ports have been reversed compared to the current packet 5-tuple. Those information will be used during the lookup process to understand if the current packet is in the same direction as the one originating the connection, or the opposite.

Lookup process. When a packet arrives to the *Conntrack Label* module, it computes the key for the current packet according to the previous formula and determines the *ip reverse* and *port reverse* as before (that we call `ipRev` and `portRev`). At this point, using this key, it performs a lookup into the *conntrack* table; if the lookup succeeds, the new flags are compared with those saved in the *conntrack* table; if they are exactly the opposite, as shown in the following formula:

```
(currIpRev != IpRev) && (currPortRev != PortRev)   (2)
```

we are dealing with the reverse packet related to that stored session. On the other hand, if they are exactly the same it means that we are dealing with a packet in the same direction as the stored one.

Stateful matching module. If at least one rule of the ruleset requires a stateful match, the `bpf-iptables` control plane instantiates an additional module within the classification pipeline, called *Conntrack Match*; this module uses a `BPF_ARRAY`, filled with all the possible value-bitvector pairs for each possible label, to find the

[6] eBPF programs have also the possibility to read the payload of the packet (e.g., [1]), which is required to recognize *related* connections. Supporting these features in `bpf-iptables` can be done by extending the conntrack module to recognize the different L7 protocol from the packet and inserting the correct information in the conntrack table.

[7] The possible labels that the conntrack module associates to a packet are the same defined by the `netfilter` framework (i.e., `NEW, ESTABLISHED, RELATED, INVALID`).

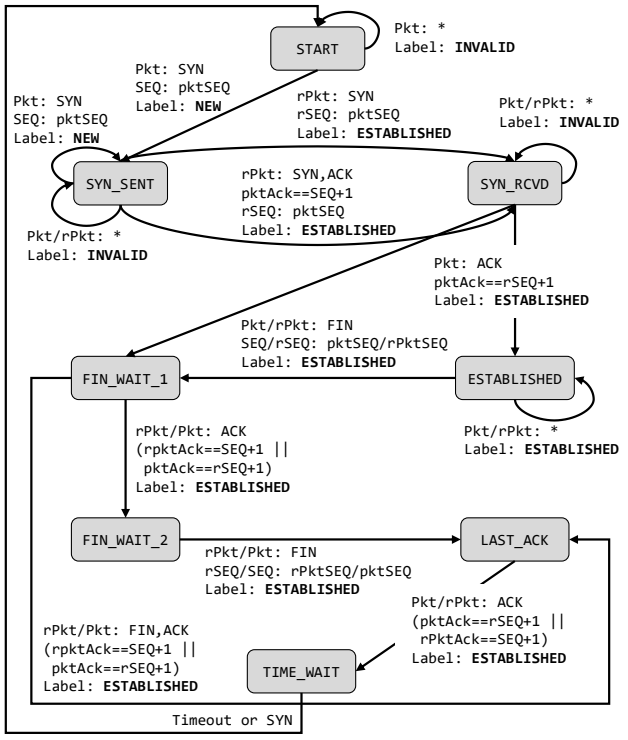


Figure 5: The TCP state machine for the `bpf-iptables` conntrack function. The grey boxes indicate the connection states saved into the conntrack table, while the label represents the value assigned by the first conntrack module before the packet enters the classification pipeline.

bitvector associated to the current label. While the *Conntrack Match* is only instantiated whenever there is a stateful rule in the ruleset, this is not true for the two connection tracking modules outside the classification pipeline, which continue to track the existing connections in order to be ready for state-based rules instantiated subsequently.

TCP state machine. To provide a better understanding of the various operations performed by the `bpf-iptables` connection tracking module and the level of complexity that is needed to support this feature, we show the TCP state machine implemented in `bpf-iptables` in Figure 5, which reveals not only the different TCP state but also the corresponding labels assigned at each step. The first state transition is triggered by a `TCP SYN` packet (all other packets not matching that condition are marked with the `INVALID` label); in this case, if the packet is accepted by the classification pipeline, the new state (i.e., `SYN_SENT`) is stored into the *conntrack* table together with some additional flow context information such as the last seen sequence number, which is used to check the packet before updating the connection state. Figure 5 refers to *forward* or *reverse* packet (i.e., `pkt` or `rPkt`) depending on the initiator of the connection. Finally, when the connection reaches the `TIME_WAIT` state, only a timeout event or a new `SYN` from the same flow will trigger a state change. In the first case the entry is deleted from the *conntrack* table, otherwise the current packet direction is marked as forward and the new state becomes `SYN_SENT`.

Conntrack Cleanup. `bpf-iptables` implements the cleanup of conntrack entries in the control plane, with a thread that checks the presence of expired sessions. For this, the `bpf-iptables` conntrack data plane updates a *timestamp* value associated with that connection when a new packet is received, which can be subsequently read by the control plane *cleanup* thread. Since we noticed that the usage of the `bpf_ktime()` helper to retrieve the current timestamp causes a non-negligible performance overhead, the *cleanup* thread updates a per-CPU array map every second with a new timestamp value, which is used by the data plane when the entry is updated. Even if this value does not perfectly indicate the *current* timestamp, we believe that it is a good trade-off between performance and accuracy for this type of application.

5 CONTROL PLANE

This Section describes the main operations performed by the control plane of `bpf-iptables`, which are triggered whenever one of the three following events occur.

`bpf-iptables` start-up. To behave as `iptables`, `bpf-iptables` has to intercept all incoming and outgoing traffic and handle it in its custom eBPF pipeline. When started, `bpf-iptables` attaches a small eBPF *redirect* program to the ingress (and egress) hook of each host's interface visible from the root namespace, as shown in Figure 2. This program intercepts all packets flowing through the interface and "jumps" into the first program of the `bpf-iptables` ingress or egress pipeline. This enables the creation of a single processing pipeline that handles all the packets, whatever interface they come from; in fact, eBPF programs attached to a NIC cannot serve multiple interfaces, while eBPF modules not directly attached to a NIC can be called from multiple eBPF starting programs. Finally, `bpf-iptables` retrieves all local IP addresses active on any NIC and configures them in the *Chain Selector*; this initialization phase is done by subscribing to any `netlink` event related to status or address changes on the host's interfaces.

`Netlink` notification. Whenever a new `netlink` notification is received, `bpf-iptables` checks if the notification is related to specific events in the root namespace, such as the creation of an interface or the update of an IP address. In the first case, the *redirect* program is attached to the eBPF hook of the new interface so that packets received from it can be intercepted by `bpf-iptables`. In the second case, the `netlink` notification indicates that the IP address of an interface has changed; as consequence, `bpf-iptables` updates the map of local IPs used in the *Chain Selector* with the new address.

Ruleset changes. Obviously, the control plane is involved when the configuration of the ruleset changes. This process involves the execution of the pre-processing algorithm, which calculates the value-bitvector pairs for each field; those values will be then inserted into the new eBPF maps and the new programs created on the parallel chain.

The **pre-processing** algorithm requires that all the tables used in the pipeline are correctly filled with the bitvector representing the current ruleset. Let's assume we have a list of N packet filtering rules that require exact or wildcard matching on a set of K fields; *(i)* for each field $k_i \in K$ we extract a set of distinct values $\theta_i =$

$\{k_{i,1}, k_{i,2}, ..., k_{i,j}\}$ with $j \leq card(N)$ from the current ruleset N; *(ii)* if there are rules that require wildcard matching for the field k_i, we add an additional entry to the set θ_i that represents the wildcard value; *(iii)* for each $k_{i,j} \in \theta_i$ we scan the entire ruleset and if $\forall n_i \in N$ we have that $k_{i,j} \subseteq n_i$ then we set the bit corresponding to the position of the rule n_i in the bitvector for the value $k_{i,j}$ to 1, otherwise we set the corresponding bit to 0. Repeating these steps for each field $k_i \in K$ will allow to construct the final value-bitvector pairs to be used in the classification pipeline. The Algorithm 1 shows the corresponding pseudo-code.

The final step for this phase is to insert the generated values in their eBPF maps. Each matching field has a default map; however, `bpf-iptables` is also able to choose the map type at runtime, based on the current ruleset values. For example, a `LPM_TRIE` is used as default map for IP addresses, which is the ideal choice when a range of IP addresses is used; however, if the current ruleset contains only rules with fixed (/32) IP addresses, it changes the map into a `HASH_TABLE`, making the matching more efficient. Before instantiating the pipeline, `bpf-iptables` modifies the behavior of every single module by regenerating and recompiling the eBPF program that best represents the current ruleset. When the most appropriate map for a given field has been chosen, `bpf-iptables` fills it with computed value-bitvector pairs. The combination of eBPF map and field type affects the way in which `bpf-iptables` represents the wildcard rule. In fact, for maps such as the `LPM_TRIE`, used to match IP addresses, the wildcard can be represented as the value `0.0.0.0/0`, which is inserted as any other value and will be matched whenever the are no other matching values. On the other hand, for L4 source and destination ports, which use a `HASH_MAP`, `bpf-iptables` instantiates the wildcard value as a variable hard-coded in the eBPF program; when the match in the table fails, it will use the wildcard variable as it was directly retrieved from the map.

`Bpf-iptables` adopts a variant of the previous algorithm for fields that have a limited number of possible values, where instead of generating the set θ_i of distinct values for the field k_i, it produces all possible combinations for that value. The advantage is that *(i)* it does not need to generate a separate bitvector for the wildcard, being all possible combinations already contained within the map and *(ii)* can be implemented with an eBPF `ARRAY_MAP`, which is faster compared to other maps. An example is the processing of TCP flags; since the number of all possible values for this field is limited (i.e., 2^8), it is more efficient to expand the entire field with all possible cases instead of computing exactly the values in use.

Algorithm 1 Pre-processing algorithm

Require: N, the list of filtering rules
Extract K, the set of matching fields used in N
for each $k_i \in K$ **do**
  $b_i \leftarrow$ # bit of field K_i
  $\theta_i \leftarrow \{k_{i,j} \mid \forall j \leq \min(card(N), 2^{b_i})\}$ ▷ set of distinct values
  if $\exists$ a wildcard rule $\in N$ for k_i **then**
    Add wildcard entry to θ_i
  for each $k_{i,j} \in \theta_i$ **do**
    $bitvector_{i,j}[N] \leftarrow \{0\}$
    for each $n_i \in N$ **do**
      if $k_{i,j} \subseteq n_i$ **then**
        $bitvector_{i,j}[i] = 1$

6 EVALUATION

This Section evaluates the performance, correctness and deployability of `bpf-iptables`. First, Section 6.1 describes our test environment, the evaluation metrics and the different ruleset and packet traces used. Next, Section 6.2 evaluates the performance and efficiency of individual `bpf-iptables` components (e.g., connection tracking, matching pipeline). Finally, Section 6.3 simulates some common use cases showing how `bpf-iptables` can bring significant benefits compared to the existing solutions.

6.1 Test environment

Setup. Our testbed includes a first server used as DUT running the firewall under test and a second used as packet generator (and possibly receiver). The DUT encompasses an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) with support for Intel's Data Direct I/O (DDIO) [23], 19.25 MB of L3 cache and two 32GB RAM modules. The packet generator is equipped with an Intel® Xeon CPU E3-1245 v5 4-cores CPU @3.50GHz (8 cores with hyper-threading), 8MB of L3 cache and two 16GB RAM modules. Both servers run Ubuntu 18.04.1 LTS, with the packet generator using kernel 4.15.0-36 and the DUT running kernel 4.19.0. Each server has a dual-port Intel XL710 40Gbps NIC, each port directly connected to the corresponding one of the other server.
Evaluation metrics. Our tests analyze both TCP and UDP throughput of `bpf-iptables` compared to existing (and commonly used) Linux tools, namely `iptables` and `nftables`. TCP tests evaluate the throughput of the system under "real" conditions, with all the offloading features commonly enabled in production environments. Instead, UDP tests stress the capability of the system in terms of packet per seconds, hence we use 64B packets without any offloading capability. When comparing `bpf-iptables` with `iptables` and `nftables` we disabled *(i)* their corresponding connection tracking modules (i.e., `nf_conntrack` and `nft_ct`), since `bpf-iptables` uses its own connection tracking, (Section 4.5), and *(ii)* the other kernel modules related to `iptables` and `nftables` (e.g., `x_tables`, `nf_tables`). Although most of the evaluation metrics are common among all tests, we provide additional details on how the evaluation has been performed on each test separately.
Testing tools. UDP tests used `Pktgen-DPDK` v3.5.6 [16] and `DPDK` v18.08 to generate traffic, while for TCP tests we used either `iperf` or `weighttp` [3] to generate a high number of *new* parallel TCP connection towards the DUT, counting only the successful completed connections [24]. Particularly, the latter reports the actual capability of the server to perform real work.
Rulesets and Packet-traces. We generated synthetic rulesets that vary depending on the test under consideration, for this reason we describe the ruleset content in the corresponding test's section. Regarding the generated traffic, we configured `Pktgen-DPDK` to generate traffic that matches the configured rules; even in this case we discuss the details in every test's description.

6.2 System benchmarking

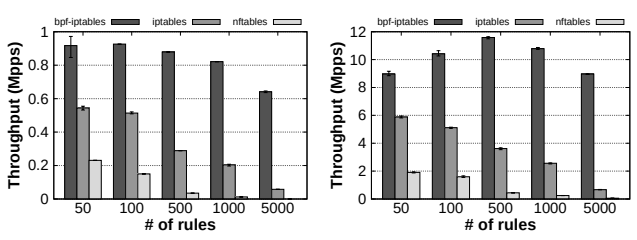


Figure 6: Single 6(a) and multi-core 6(b) comparison when increasing the number of loaded rules. Generated traffic (64B UDP packets) is uniformly distributed among all the rules.

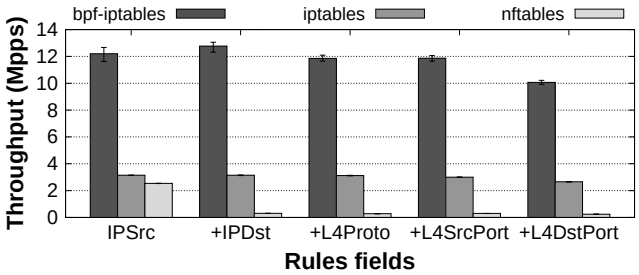


Figure 7: Multi-core performance comparison when varying the number of fields in the rulesets. Generated traffic (64B UDP packets) is uniformly distributed among all the rules.

6.2.1 Performance dependency on the number of rules. This test evaluates the performance of `bpf-iptables` with an increasing number of rules, from 50 to 5k.

Ruleset. We generated five synthetic rulesets with all the rules matching the 5-tuple of a given packet. Rules have been loaded in the `FORWARD` chain and the DUT has been configured as router in order to forward all traffic received on the first interface to the second. `nftables` rules have been generated using the same rules loaded for `bpf-iptables` and `iptables` but converted using the `iptables-translate` tool [5].

Test setup. The packet generator is configured to generate traffic uniformly distributed among all the rules[8] so that all packets will uniformly match the rules of the ruleset and no packet will match the default action of the chain, in other words, the number of flows generated is equal to the number of rules under consideration.

Evaluation metrics. We report the UDP throughput (in Mpps) among 10 different runs. This value is taken by adjusting the rate at which packets are sent in order to find the maximum rate that achieves a packet loss less than 1%. Single-core results are taken by setting the interrupts mask of each ingress receive queue to a single core, while multi-core performance represent the standard case where all the cores available in the DUT are used.

Results. Figure 6(a) and 6(b) show respectively the single-core and multi-core forwarding performance results for this test. We can notice from Figure 6(a) how `bpf-iptables` outperforms `iptables` by a factor of two even with a relatively small number of rules (i.e., 50) and this gap is even larger when considering `nftables` of which `bpf-iptables` is even almost 5 times better with 50 rules. When the number of rules grows, the performance advantage of `bpf-iptables` is more evident thanks to its improved classification pipeline, although its performance decreases as well when a large number of rules are loaded. This is due to the necessity to scan the entire bitvector in order to find the final matching rule (Section 4.4). Finally, Figure 6(b) shows how `bpf-iptables` scale across multiple cores; the maximum throughput is achieved with 1K rules, which is due to the fact that with a smaller number of rules the number of flows generated by the packet generator is not enough to guarantee uniform processing across multiple cores (due to the RSS/RFS feature of the NIC), with a resulting lower throughput.

6.2.2 Performance dependency on the number of matching fields. Since the `bpf-iptables` modular pipeline requires a separate eBPF program (hence an additional processing penalty) for each matching field, this test evaluates the throughput of `bpf-iptables` when augmenting the number of matching fields in the deployed rules in order to characterize the (possible) performance degradation when operating on a growing number of protocol fields.

Ruleset. We generated five different rulesets with a fixed number of rules (i.e., 1000) and with an increasing complexity that goes from matching only the srcIP address to the entire 5-tuple. All the rules have been loaded in the `FORWARD` chain and have the `ACCEPT` action, while the default action of the chain is `DROP`. As before, `nftables` rules have been generated using `iptables-translate` tool.

Test setup and evaluation metrics. Same as Section 6.2.1.

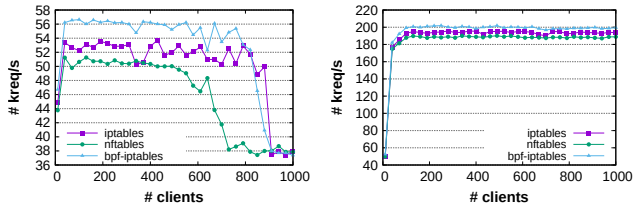
Results. Results (Figure 7) show that `iptables` performs almost the same independently on the complexity of the rules; this is expected given that is cost is dominated by the number of rules. Results for `bpf-iptables` are less obvious. While, in the general case, increasing the number of fields corresponds to a decrease in performance (e.g., rules operating on the 5-tuple show the lowest throughput), this is not always true, with the first four columns showing roughly the same value and the peak observed when operating on two fields. In fact, the performance of `bpf-iptables` are influenced also by the *type* of field and *number* of values for each field. For instance, the matching against IP addresses requires, in the general case, a longest prefix match algorithm; as consequence, `bpf-iptables` uses an `LPM_TRIE`, whose performance are dependent on the number of distinct values. In this case, a single matching on a bigger LPM_TRIE results more expensive than two matches on two far smaller LPM_TRIE, which is the case when rules operate on both IP source and destination addresses[9].

6.2.3 Connection Tracking Performance. This test evaluates the performance of the connection tracking module, which is required to enable stateful filtering. This test was based on TCP traffic in order to stress the rather complex state machine of the TCP protocol (Section 4.5), which was achieved by generating a high number of *new* connections per second, taking the number of successfully completed sessions as performance indicator.

Test setup. In this test `weighttp` [3] generated 1M HTTP requests towards the DUT, using an increasing number of concurrent clients

[8]We used a customized version of Pktgen-DPDK [27] to randomly generate packet for a given range of IP addresses and L4 port values.

[9]First ruleset had 1000 rules, all operating on source IP addresses. Second ruleset used #50 distinct srcIPs and #20 distinct dstIPs, resulting again in 1000 rules.



(a) NIC interrupts set to a single core; `nginx` running on the remaining ones.

(b) NIC interrupts are set to all the cores; `nginx` running without any restrictions.

Figure 8: Connection tracking with an increasing number of clients (number of successfully completed requests/s).

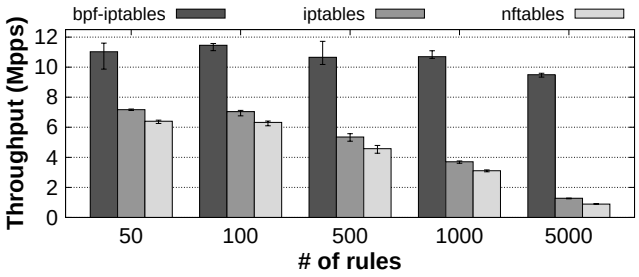


Figure 9: Throughput when protecting a variable number of services within a DMZ. Multi-core tests with UDP 64B packets, bidirectional flows.

to stress the connection tracking module. At each request, a file of 100 byte is returned by the `nginx` web server running in the DUT. Once the request is completed, the current connection is closed and a new connection is created. This required to increase the limit of 1024 open file descriptors per process imposed by Linux in order to allow the sender to generate a larger number of new requests per second and to enable the `net.ipv4.tcp_tw_reuse` flag to reuse sessions in `TIME_WAIT` state in both sender and receiver machines[10].

Ruleset. The ruleset used in this test is composed of three rules loaded in the `INPUT` chain so that only packets directed to a local application will be processed by the firewall. The first rule *accepts* all packets belonging to an `ESTABLISHED` session, the second rule *accepts* all the `NEW` packets coming from the packet generator and with the TCP destination port equal to 80 and finally, the last rule *drops* all the other packets coming from the packet generator.

Evaluation metrics. We measure the number of successfully completed requests; in particular, `weighttp` increments the number of successfully completed requests if it is completed within 5 seconds, otherwise a failure is recorded.

Results. `bpf-iptables` achieves a higher number of requests per second in both single-core and multi-core tests, with `iptables` performing from 5 to 3% less and `nftables` being down from 7 to 10%, as shown in Figures 8(a) and 8(b). However, for the sake of precision, the connection tracking module of `bpf-iptables` does not include all the features supported by `iptables` and `nftables` (Section 4.5). Nevertheless, we remind that this logic can be customized at run-time to fit the necessity of the particular running application, including only the required features, without having to update the existing kernel module.

6.3 Common use cases

In this set of tests we analyzed some scenarios that are common in enterprise environments, such as *(i)* protecting servers in a DMZ, and *(ii)* performance under DDoS attack.

6.3.1 Enterprise public servers. This test mimics the configuration of an enterprise firewall used as *front-end* device, which controls the traffic directed to a protected network (e.g., `DMZ`) that hosts a set of servers that must be reachable from the outside world. We increase the number of public servers that needs to be protected, hence tests were repeated with different number of rules.

Ruleset. The first rule *accepts* all the `ESTABLISHED` connections towards the protected network; then, a set of rules *accept* `NEW` connections generated by the servers in the protected network toward the outside world; the latest set of rules enable the communication towards the services exposed in the protected network by matching on the destination IP, protocol and L4 port destination of the incoming packets. Among the different runs we used an increasing number of rules ranging from 50 to 5K, depending on the number of public services that are exposed to the outside world.

Test setup. All the rules are loaded in the `FORWARD` chain and the traffic is generated so that the 90% is evenly distributed among all the rules and the 10% matches the default `DROP` rule. The packet generator is connected to the DUT through two interfaces, simulating a scenario where the firewall is in between the two (public and protected) networks. In particular, the first interface simulates the traffic coming from the external network i.e., a set of clients contacting the internal services, while the second interface simulates a response from the internal services to the clients. For this reason, during this test, when the traffic coming from the external and the internal network reaches the firewall, it considers all the connection as `ESTABLISHED`, hence matching the first rule of the ruleset, which represents a common scenario in an enterprise network.

Evaluation metrics. The test has been repeated 10 times with the results reporting the throughput in Mpps (for 64B UDP packets) retrieved by looking at the number of packets processed by the firewall and dividing it by the test duration.

Results. `bpf-iptables` outperforms existing solutions thanks to the optimized path for the `FORWARD` chain, which transparently avoids the overhead of the Linux TCP/IP stack, as shown in Figure 9. In addition, its throughput is almost independent from the number of rules thanks to the optimization on the `ESTABLISHED` connections (Section 4.4.2), which avoids the overhead of the classification pipeline if the conntrack module recognizes an `ESTABLISHED` connection that should be *accepted*. Even if `iptables` would also benefit from the fact that most packets match the first rule, hence making the linear search faster, the overall performance in Figure 9 show a decrease in throughput when the number of rules in the ruleset grows. This is primarily due to the overhead to recognize the traffic matching the default rule (`DROP` in our scenario), which still requires to scan (linearly) the entire ruleset.

[10] We also tuned some parameters (e.g., max backlog, local port range) in order to reduce the overhead of the web server.

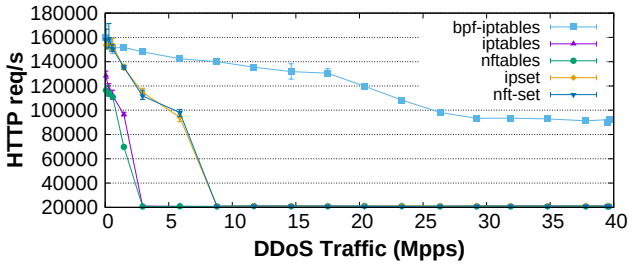


Figure 10: Multi-core performance under DDoS attack. Number of successful HTTP requests/s under different load rates.

6.3.2 Performance under DDoS Attack. This tests evaluates the performance of the system under DDoS attack. We analyzed also two optimized configurations of `iptables` and `nftables` that make use of `ipset` and `sets` commands, which ensures better performance when matching an entry against a set of values.

Ruleset. We used a fixed set of rules (i.e., 1000) matching on IP source, protocol and L4 source port, `DROP` action. Two additional rules involve the connection tracking to guarantee the reachability of internal servers; (*i*) *accepts* all the `ESTABLISHED` connections and (*ii*) *accepts* all the `NEW` connection with destination L4 port 80.

Test setup and evaluation metrics. The packet generator sends 64Bytes UDP packets towards the server with the same set of source IP addresses and L4 ports configured in the set of blacklisting rules. DDoS traffic is sent on a first port connected to the DUT, while a `weighttp` client sends traffic on a second port, simulating a legitimate traffic towards a `nginx` server running in the DUT. `Weighttp` generates 1M HTTP requests using 1000 concurrent clients; we report the number of successfully completed requests/s, with a timeout of 5 seconds, varying the rate of DDoS traffic.

Results. Results, depicted in Figure 10 (multicore scenario), show that the performance of `bpf-iptables`, `ipset` and `nft-set` are comparable in case of low-volume DDoS attacks; `iptables` and `nftables` are slightly worse because of their matching algorithm that suffers with an high number of rules. However, with higher DDoS load (> 8Mpps), also the performance of `ipset` and `nft-set` drop rapidly and the server becomes unresponsive, with almost no requests/s served; `iptables` and `nftables` are even worse (zero goodput at 2.5Mpps). Vice versa, thanks to its matching pipeline at the XDP level, `bpf-iptables` is able to successfully sustain about 95.000 HTTP requests/s (i.e., about 60% of the maximum achievable load) of legitimate traffic when the DDoS attack rate is more than 40Mpps. Higher DDoS load was not tested because of a limitation of the traffic generator in use.

6.4 Microbenchmarks

We now provide a set of micro-benchmarks that characterize the system under specific conditions.

6.4.1 Baseline performance. This test analyzes the overhead of `bpf-iptables` on a vanilla system, without any firewall rule. This condition still requires the presence of some processing modules such as connection tracking and the logic that applies the default action (i.e., `ALLOW`) to all packets. This represents the most favorable case for `iptables` where cost grows linearly with the number of rules, while `bpf-iptables` has to pay the cost of some programs at the beginning of the pipeline that must be always active.

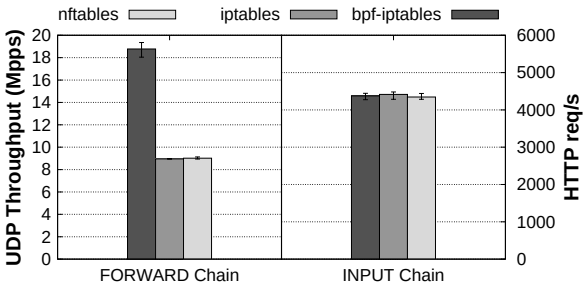


Figure 11: Performance with single default `ACCEPT` rule (baseline). Left: UDP traffic, 64B packets matching the `FORWARD` chain. Right: number of HTTP requests/s (downloading a 1*MB* web page), TCP packets matching the `INPUT` chain.

Table 1: Comparison of the time required to append the $(n+1)^{th}$ in the ruleset (ms).

# rules	ipt	nft	bpf-iptables			HORUS	
			t1[1]	t2[2]	t3[3]	t_H1[4]	t_H2[5]
0	15	31	0.15	1165	0.34	382	0.0024
50	15	34	2.53	1560	0.36	1.08	0.0026
100	15	35	5.8	1925	0.35	2.06	0.0026
500	16	36	17	1902	0.34	8.60	0.0027
1000	17	69	33.4	1942	0.34	14.4	0.0027
5000	28	75	135	2462	0.38	37.3	0.0031

[1] Time required to compute all the bitvectors-pairs.
[2] Time required to create and load the new chain.
[3] Time required to remove the old chain.
[4] Time required to identify the rules belonging to a `HORUS` set.
[5] Time required to insert the new rule in the `HORUS` set.

The right side of Figure 11 shows the performance of `bpf-iptables`, `iptables` and `nftables` when the traffic (64B UDP packets) traverses the `FORWARD` chain. This case shows a considerable advantage of `bpf-iptables` thanks to its optimized forwarding mechanism (Section 4.4.2). The situation is slightly different when the traffic hits the `INPUT` chain (Figure 11, right). In fact, in such case the packets has to follow the usual path towards the stack before reaching the local application, with no chance to shorten its journey. While `bpf-iptables` does not show the improvement seen in the previous case, it does not show any worsening either, hence demonstrating that the active code (e.g., the new *conntrack* implementation) introduces the same overhead of other solutions.

6.4.2 Rules insertion time. The LBVS matching algorithm requires the update of the entire pipeline each time the ruleset changes (Section 4.4.3). This test evaluates the time required to insert the $(n+1)^{th}$ rule when the ruleset already contains n rules; in case of `iptables` and `nft`, this has been measured by computing the time required to execute the corresponding userspace tool. Results, presented in Table 1, show that both `iptables` and `nftables` are very

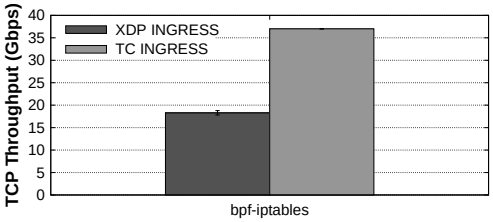


Figure 12: TCP throughput when the `bpf-iptables` ingress pipeline (with zero rules) is executed on either `XDP` or `TC` ingress hook; `bpf-iptables` running on a single CPU core; `iperf` running on all the other cores.

fast in this operation, which completes in some tens of milliseconds; `bpf-iptables`, instead, requires a far larger time (varying from 1 to 2.5s with larger rulesets). To understand the reason of this higher cost, we exploded the `bpf-iptables` rules insertion time in three different parts. Hence, `t1` indicates the time required by the `bpf-iptables` control plane to compute all the value-bitvector pairs for the current ruleset. Instead, `t2` indicates the time required to load the new parallel chain (i.e., compile and load the new eBPF classification pipeline); during this time, `bpf-iptables` continues to process the traffic according to the old ruleset, with the *swapping* performed as last step when all the new modules are successfully created and injected in the kernel[11]. Finally, `t3` is the time required to delete the old chain, which does not really impact on user experience since the new pipeline is already filtering traffic after `t2`.

Finally, the last column of Table 1 depicts the time required by `bpf-iptables` to insert a rule that can be handled by `HORUS` (Section 4.4.2). Excluding the first entry of the `HORUS` set that requires to load the `HORUS` eBPF program, all the other entries are inserted in the `HORUS` set within an almost negligible amount of time (t_H2). Instead, the detection if the new rule belongs to an `HORUS` set takes more time (t_H1 ranges from 1 to 40ms), but this can be definitely reduced with a more optimized algorithm.

6.4.3 Ingress pipeline: XDP vs. TC. `bpf-iptables` attaches its ingress pipeline on the `XDP` hook, which enables traffic processing as early as possible in the Linux networking stack. This is particularly convenient when the packet matches the `DROP` action or when there is the possibility to bypass the TCP/IP stack and forward immediately the packet to the final destination (*optimized forwarding* in Section 4.4.2). However, when an eBPF program is attached to the `XDP` hook, the Generic Receive Offload[12] feature on that interface is disabled; as a consequence, we may incur in higher processing costs in presence of large TCP *incoming* flows.

Results in Figure 12, which refer to a set of parallel TCP flows between the traffic generator and the DUT, with a void `INPUT` chain and the default `ACCEPT` action, show clearly how the `XDP` ingress pipeline pays a higher cost compared to `TC`, which easily saturates the 40Gbps bandwidth of the link[13]. This higher cost is given by the larger number of (small) packets to be processed by `bpf-iptables` because of the lack of GRO aggregation; it is important to note that this cost is not present if TCP data exits from the server (*outgoing* traffic), which is in fact a more common scenario.

[11] Since time `t2` depends on the number of matching fields required by each rule (`bpf-iptables` instantiates the minimum set of eBPF programs necessary to handle the current configuration), numbers in Table 1 take into account the worst case where all the rules require matching on all the supported fields.

[12] Generic Receive Offload (GRO) is a software-based offloading technique that reduces the per-packet processing overhead by reassembling small packets into larger ones.

7 CONCLUSIONS

This paper presents `bpf-iptables`, an eBPF based Linux firewall designed to preserve the `iptables` filtering semantic while improving its speed and scalability, in particular when a high number of rules are used. Being based on eBPF, `bpf-iptables` is able to take advantage of the characteristics of this technology, such as the dynamic compilation and injection of the eBPF programs in the kernel at run-time in order to build an optimized data-path based on the actual firewall configuration. The tight integration of `bpf-iptables` with the Linux kernel may represent a great advantage over other solutions (e.g., DPDK) because of the possibility to cooperate with the rest of the kernel functions (e.g., routing) and the other tools of the Linux ecosystem. Furthermore, `bpf-iptables` does not require custom kernel modules or additional software frameworks that could not be allowed in some scenarios such as public data-centers.

`Bpf-iptables` guarantees a huge performance advantage compared to existing solutions, particularly in case of an high number of rules; furthermore, it does not introduce undue overheads in the system when no rules are instantiated, even though in some cases the use of `XDP` on the ingress hook could hurt the overall performance of the system. Existing eBPF limitations have been circumvented with ad-hoc engineering choices (e.g., classification pipeline) and clever optimizations (e.g., HORUS), which guarantee further scalability and fast update time.

On the other hand, currently `bpf-iptables` supports only a subset of the features available in `Netfilter`-based firewalls. For instance, `iptables` is often used to also handle *natting* functions, which we have not considered in this paper, as well as the features available in `ebtables` and `arptables`. Those functionality, together with the support for additional matching fields are considered as possible direction for our future work.

8 ACKNOWLEDGEMENT

Authors would like to thank the many people who contributed to this work, among the others Pere Monclus, Aasif Shaikh and Massimo Tumolo. Our thanks also to Vmware, which partially funded this project.

REFERENCES

[1] BCC Authors. 2016. HTTP Filter. https://github.com/iovisor/bcc/tree/master/examples/networking/http_filter [Online; last-retrieved 15-November-2018].
[2] Cilium Authors. 2018. BPF and XDP Reference Guide. https://cilium.readthedocs.io/en/latest/bpf/
[3] Lighttpd authors. 2018. weighttp: a lightweight and simple webserver benchmarking tool. https://redmine.lighttpd.net/projects/weighttp/wiki [Online; last-retrieved 10-November-2018].
[4] Nftables authors. 2016. Main differences with iptables. https://wiki.nftables.org/wiki-nftables/index.php/Main_differences_with_iptables

[13] To avoid TCP and application-level processing to become the bottleneck, we set all the NIC interrupts to a single CPU core, on which `bpf-iptables` has to be executed, while `iperf` uses all the remaining ones.

[5] Netfilter Authors. 2018. Moving from iptables to nftables. https://wiki.nftables.org/wiki-nftables/index.php/Moving_from_iptables_to_nftables [Online; last-retrieved 10-October-2018].

[6] Pablo Neira Ayuso. 2018. [PATCH RFC PoC 0/3] nftables meets bpf. https://www.mail-archive.com/netdev@vger.kernel.org/msg217425.html

[7] David Beckett. 2018. Hello XDP_DROP. https://www.netronome.com/blog/hello-xdp_drop/ [Online; last-retrieved 15-November-2018].

[8] D. Borkmann. 2018. net: add bpfilter. https://lwn.net/Articles/747504/ [Online; last-retrieved 30-June-2018].

[9] Jesper Dangaard Brouer. 2018. XDP Drivers. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/drivers.html [Online; last-retrieved 18-September-2018].

[10] Jesper Dangaard Brouer and Toke Høiland-Jørgensen. 2003. XDP: challenges and future work. In *LPC'18 Networking Track*. Linux Plumbers Conference.

[11] J. Corbet. 2009. Nftables: a new packet filtering engine. https://lwn.net/Articles/324989 [Online; last-retrieved 30-June-2018].

[12] Jonathan Corbet. 2018. BPF comes to firewalls. https://lwn.net/Articles/747551/

[13] Edward Cree. 2018. Bounded loop support work in progress. https://lwn.net/Articles/748032/

[14] Edward Cree. 2018. Bounded Loops for eBPF. https://lwn.net/Articles/756284/

[15] James Daly and Eric Torng. 2017. TupleMerge: Building Online Packet Classifiers by Omitting Bits. In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*. IEEE, 1–10.

[16] DPDK. 2018. Pktgen Traffic Generator Using DPDK. http://dpdk.org/git/apps/pktgen-dpdk

[17] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/

[18] T. Graf. 2018. Why is the kernel community replacing iptables with BPF? https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables [Online; last-retrieved 30-June-2018].

[19] Pankaj Gupta and Nick McKeown. 1999. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, Vol. 40.

[20] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *CoNEXT'18: International Conference on emerging Networking EXperiments and Technologies*. ACM Digital Library.

[21] Docker Inc. 2018. Docker. https://www.docker.com/ [Online; last-retrieved 30-June-2018].

[22] Facebook Inc. 2018. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/ [Online; last-retrieved 30-June-2018].

[23] Intel(R). 2018. IntelÂő Data Direct I/O Technology. https://www.intel.it/content/www/it/it/io/data-direct-i-o-technology.html [Online; last-retrieved 09-November-2018].

[24] József Kadlecsik and György Pásztor. 2004. Netfilter performance testing. (2004).

[25] 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.

[26] Charles E Leiserson, Harald Prokop, and Keith H Randall. 1998. Using de Bruijn sequences to index a 1 in a computer word. *Available on the Internet from http://supertech. csail. mit. edu/papers. html* 3 (1998), 5.

[27] Sebastiano Miano. 2018. Custom Pktgen-DPDK version. https://github.com/sebymiano/pktgen-dpdk

[28] 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.

[29] Thomas Heinz Michael Bellion. 2002. NF-HIPAC: High Performance Packet Classification for Netfilter. https://lwn.net/Articles/10951/

[30] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. 2009. Packet classification algorithms: From theory to practice. In *INFOCOM 2009, IEEE*. IEEE, 648–656.

[31] P. Russell. 1998. The netfilter.org project. https://netfilter.org/ [Online; last-retrieved 30-June-2018].

[32] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 213–224.

[33] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, Vol. 29. ACM, 135–146.

[34] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. 1998. *Fast and scalable layer four switching*. Vol. 28. ACM.

[35] Alexei Starovoitov. 2014. *net: filter: rework/optimize internal BPF interpreter's instruction set.* In Linux Kernel, commit bd4cf0ed331a.

[36] Balajee Vamanan, Gwendolyn Voskuilen, and TN Vijaykumar. 2011. EffiCuts: optimizing packet classification for memory and throughput. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 207–218.

[37] Nic Viljoen. 2018. BPF, eBPF, XDP and Bpfilter...What are These Things and What do They Mean for the Enterprise? https://goo.gl/GHaJTz [Online; last-retrieved 15-November-2018].

[38] J. Wallen. 2015. An Introduction to Uncomplicated Firewall (UFW). https://www.linux.com/learn/introduction-uncomplicated-firewall-ufw [Online; last-retrieved 30-June-2018].

A APPENDIX

We aim at making our `bpf-iptables` prototype available so that anyone can use it and experiment the power of our eBPF based firewall. In this respect, it is worth remembering that the performance characterization requires a careful prepared setup, including traffic generators and proper hardware devices (server machines, NICs).

To facilitate the access and execution of `bpf-iptables`, we created a Docker image containing all the instructions necessary to run the executable, which can be used to replicate the results described in this paper. The Docker image is hosted on a DockerHub repository and can be downloaded with the following command:

```
$ docker pull netgrouppolito/bpf-iptables:latest
```

Once downloaded, the image can be executed with the underlying command, which will print a detailed description on the terminal containing all the information necessary to execute `bpf-iptables` and how to use it.

```
$ docker run -it netgrouppolito/bpf-iptables
```

Rulesets. The rulesets and the scripts used for the evaluation are provided also shipped inside the Docker image and can be found inside the direction `tests` of the container.

Source code. This software project is currently in the final testing phases and it is expected to be released in the open-source domain in the upcoming month of January 2019.