Securing Linux with a Faster and Scalable Iptables

Securing Linux with a Faster and Scalable Iptables

Sebastiano Miano
Politecnico di Torino, Italy
sebastiano.miano@polito.it

Matteo Bertrone
Politecnico di Torino, Italy
matteo.bertrone@polito.it

Fulvio Risso
Politecnico di Torino, Italy
fulvio.risso@polito.it

Mauricio Vásquez Bernal
Politecnico di Torino, Italy
mauricio.vasquez@polito.it

Yunsong Lu
Futurewei Technologies, Inc.
yunsong.lu@futurewei.com

Jianwen Pi
jianwpi@gmail.com

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 [17].

`Nftables` [10] 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. Although this yields advantages over its predecessor, `nftables` (and other previous attempts such as `ufw` [40] 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 into the new one [12]. This is also highlighted by the fact that the majority of today's open-source orchestrators (e.g., Kubernetes [21], Docker [20]) are strongly based on `iptables`.

Recently, another in-kernel virtual machine has been proposed, the extended BPF (eBPF) [2, 16, 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) [19] 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 [5]. 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` [7] 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 shows the advantages of intercepting (and therefore filtering) the traffic as soon as possible in the kernel, and even in the hardware (smartNICs) [39].

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 Linux firewall, called **bpf-iptables**, which implements an alternative filtering architecture in eBPF, while maintaining the same `iptables` filtering semantic but with improved performance and scalability. `bpf-iptables` leverages 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 [6].

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.

This paper presents the challenges, design choices and implementation of `bpf-iptables` and it compares with existing solutions such as `iptables` and `nftables`. 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 AND ASSUMPTIONS

This Section introduces *(i)* the main challenges encountered while designing `bpf-iptables`, mainly derived from the necessity to emulate the `iptables` behavior with eBPF, and *(ii)* our initial assumptions for this work, which influenced some design decisions.

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

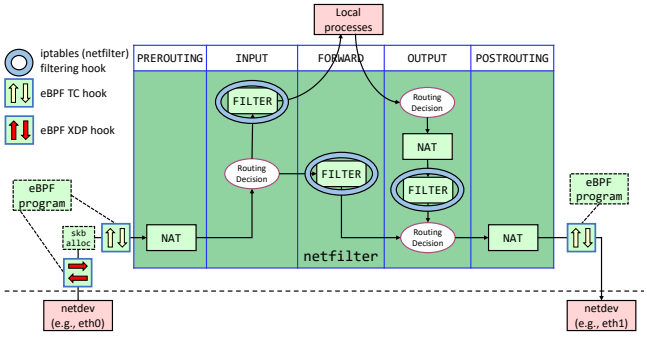


Figure 1: Location of `netfilter` and eBPF hooks.

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 [13, 18, 30, 32, 33, 38]), 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 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 support stateful rules that accept/drop packets based on the characteristic of the connection they belong to. For instance, `iptables` may accept 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),

[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.

while connections initiated from the outside world may be forbidden. As shown in Figure 1, bpf-iptables operates *before* packets enter in netfilter; being unable to exploit the Linux *conntrack* module to classify the traffic, it has to implement its own equivalent component (Section 4.5.)

2.4 Working with upstream Linux kernel

Our initial assumption for this work is to operate with the existing Linux kernel in order to bring the advantages of bpf-iptables to the wider audience as soon as possible. In fact, the process required by the Linux kernel community to agree with any non-trivial code change and have them available in a mainline kernel is rather long and may easily require more than one year. This assumption influenced, in some cases, the design choices taken within bpf-iptables (e.g., the definition of a new conntrack in eBPF instead of relying on the existing Linux one); we further analyze this point in Section 7, providing a discussion of the possible modification to the eBPF subsystem that could further improve bpf-iptables.

3 OVERALL ARCHITECTURE

Figure 2 shows the overall system architecture of bpf-iptables. The data plane includes four main classes of eBPF programs. The first set (blue) implements the classification pipeline, i.e., the ingress, forward or output chain; a second set (yellow) implements the logic required to preserve the semantics of iptables; a third set (orange) is dedicated to connection tracking. Additional programs (grey) are devoted 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 works in XDP mode, attaching all the eBPF programs to the XDP hook of the host's interfaces. However, this requires the explicit support for XDP in the NIC drivers[2]; bpf-iptables automatically falls back to the TC mode when the NIC drivers are not XDP-compatible. In the latter case, all the eBPF programs composing the *ingress* pipeline are attached to the TC hook. The *egress* pipeline is instead called upon receiving a packet on the TC egress hook, before the packet leaves the host, as XDP is not available in egress [9].

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 it to the Linux networking stack (more details 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, where it will be processed by the OUTPUT chain.

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.

[2]NIC driver with native support for XDP can be found at [8].

4 DATA PLANE

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

4.1 Header Parser

The bpf-iptables ingress and egress pipelines 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 in the data plane and has to classify and forward the traffic to the correct classification pipeline (i.e., chain) in order to preserve the iptables semantic (Section 2.1). In particular, 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 *non-local* 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 checks if the *destination* IP address of the incoming packet is present in the BPF_HASH map that keeps local IPs and writes the resulting target chain in the *packet metadata* per-CPU map shared across the entire pipeline. This value is used by the first conntrack module to jump to the correct target chain; in addition, it optimizes the (very common) case in which the first rule of the INPUT chain *accepts* all the ESTABLISHED connections by jumping directly at the end of the classification pipeline, without further processing (Section 4.4.2). 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. In fact, traffic traversing the FORWARD chain has already been matched in the ingress pipeline, hence it should not be handled by the OUTPUT chain.

4.3 Matching algorithm

To overcome the performance penalties of the linear search of iptables, bpf-iptables adopts the more efficient Linear Bit-Vector

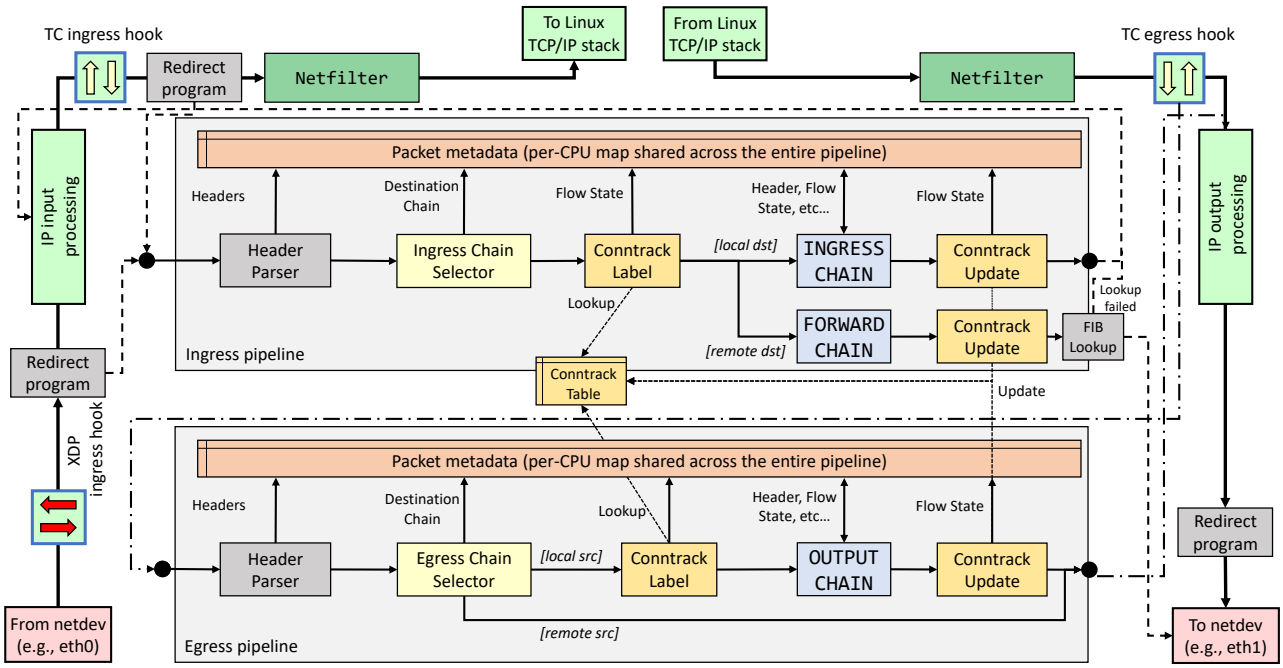


Figure 2: High-level architecture of `bpf-iptables`.

Search (LBVS) [24] 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 enabling the optimizations presented in Section 4.4.2. The algorithm follows the *divide-and-conquer* paradigm: it splits filtering rules in multiple classification steps, based on the number of protocol fields 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 fieldsshown 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 may be matched when the field assumes that value. Filtering rules, and the corresponding bits in the above bitvector, are ordered with highest priority rule first. The matching process is repeated for each field we operate 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.

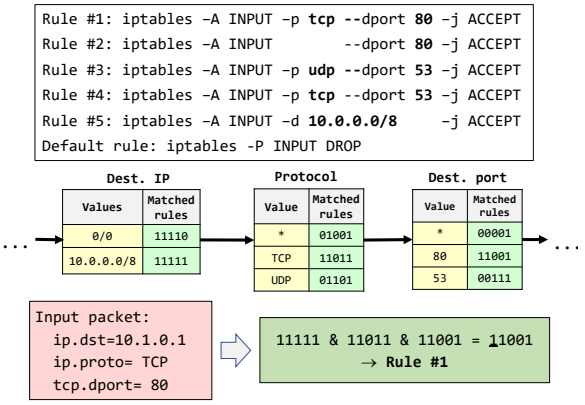


Figure 3: Linear Bit Vector Search

4.4 Classification Pipeline

The `bpf-iptables` classification pipeline (Figure 4) is in charge of filtering packets according to the rules configured for a given chain. It is made by a sequence of eBPF programs, each one handling a single matching protocol field of the current ruleset. The pipeline contains two per-CPU shared maps that keep some common information among all the programs, 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.

Each module of the pipeline performs the following operations: *(i)* extracts the needed packet fields from the *packet metadata* map, previously filled by the *Header Parser* module; *(ii)* performs a lookup on its private eBPF map to find the bitvector associated to the

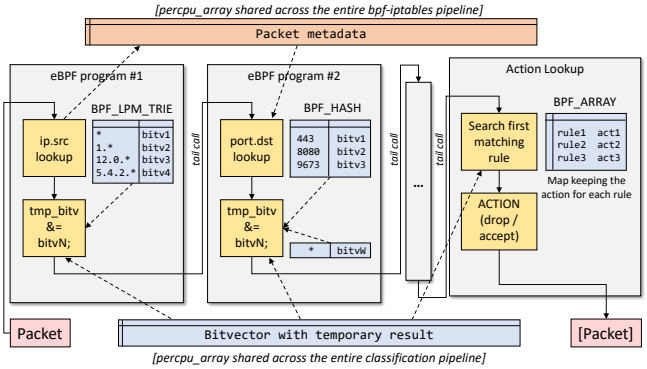


Figure 4: `bpf-iptables` classification pipeline.

current packet value for that field. If the lookup succeeds, *(iii-a)* it 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 no wildcard rules for that field, we can immediately conclude that the current packet does not match any rule within the ruleset; hence, we can exploit this situation for an early break of the pipeline (Section 4.4.2). Finally, except the last case, *(iv)* it saves the new bitvector in the shared map and calls the next module of the chain.

Bitvectors comparison. Since each 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., 2.048 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 maximum number of supported rules. The necessity to 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 [25] 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 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 pipeline and the possibility to re-generate 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 value, 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 the empty set (all bits set to 0). In either circumstance, the module that detects this situation can jump out of the pipeline by applying the default policy for the chain, without the additional overhead of executing all the following components. 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 immediately 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 matching 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. 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; putting those modules at the beginning of the pipeline could speed up processing, avoiding unnecessary memory accesses and modules.

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 updating time since we need to re-compute all the bitvectors used in the classification pipeline when the user updates the rulesets. 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 for an additional chance of early-break. In addition, since those rules are independent from the others in the ruleset, we could compact all their corresponding bits in the bitvectors with just one, hence increasing the space for other non-HORUS rules. Finally, 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; if the action is ACCEPT, it will directly jump to the last module of the pipeline, the Conntrack Update. Finally, if no match is found, HORUS jumps to the first program of the classification pipeline, following the usual processing path. An important scenario where HORUS shows its great advantages is under **DoS attacks**. In fact, if all the rules of the HORUS ruleset contains a DROP action, matching packets will be immediately discarded, 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., SmartNICs) that support the offloading of "simple" eBPF programs, further reducing the system load and the resource consumption.

Optimized forwarding. If the final decision for a packet traversing the FORWARD chain is ACCEPT, it has to be forwarded to the next-hop, according to the routing table of the host. Since, starting from kernel version 4.18, eBPF programs can query directly the Linux routing table, bpf-iptables can optimize the path of the above packet by directly forwarding the packet to the target NIC, shortening its route within the Linux stack, with a significant performance advantage (Section 6). In the (few) cases in which the needed information are not available (e.g., because the MAC address of the next hop is not yet known), 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 characteristics of the LBVS classifier is that, whenever a new rule is added, updated or removed, it needs to re-compute all the bitvectors associated with the current fields. However, to avoid inconsistency problems, we must update *atomically* the content of *all* maps in the pipeline. Unfortunately, eBPF allows the atomic update of a *single* map, while it does not support atomic updates of multiple maps. Furthermore, defining 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 exploits the fact that the classification pipeline is stateless and therefore it creates a new chain of eBPF programs and maps in parallel, based on the new ruleset. While this new pipeline is assembled and injected in the kernel, packets continue to be processed in the initial matching pipeline, accessing to the current state and configuration; when this reloading phase is completed, the Chain Selector is updated to jump to the first program of the new chain, allowing new packets to flow through it. This operation is performed *atomically*, enabling the continuous processing of the traffic with a consistent state and without any service disruption, thanks to a property of the eBPF subsystem that uses a particular map (BPF_PROG_ARRAY) to keep the addresses of the instantiated eBPF programs. Finally, when the new chain is up and running, the old one is unloaded. We discuss and evaluate the performance of the rules update 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 modules share the same BPF_HASH *conntrack* map, as shown in Figure 2.

To properly update the state of a connection, the bpf-iptables conntrack 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, outgoing packets have to be processed, in any case, by the conntrack modules located in the egress pipeline. The bpf-iptables connection tracking supports TCP, UDP, and ICMP traffic, although it does not handle advanced features such as *related* connections (e.g., when a SIP control session triggers the establishment of voice/video RTP flows[3]), nor it supports IP reassembly.

Packet walkthrough. The *Conntrack Label* module is used to associate a label to the current packet[4] by detecting any possible change

[3] eBPF programs can 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.

[4] 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).

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 to filter the packet according to the stateful rules of the ruleset. Finally, if the packet "survives" the classification pipeline, the second conntrack program (*Conntrack Update*) updates the *conntrack* table with the new connection state or, in the case of a new connection, it creates the new associated entry. Since no changes occur if the packet is dropped, 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/dst IP address, L4 protocol, src/dst L4 port) as key in the *conntrack* table. Before saving the entry in the table, the *Conntrack Update* orders the key as follows:

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

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* flags as before. At this point it performs a lookup into the *conntrack* table with this key; if the lookup succeeds, the new flags are compared with those saved in the *conntrack* table to detect which direction the packet belongs to. For instance, if:

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

we are dealing with the reverse packet related to the stored session; this is used, e.g., to mark an existing TCP session as ESTABLISHED, i.e., update its state from SYN_SENT to SYN_RCVD (Figure 5).

Stateful matching module. If at least one rule of the ruleset requires a stateful match, `bpf-iptables` instantiates also the *Conntrack Match* module within the classification pipeline to find the bitvector associated to the current label. While this module is present only when the ruleset contains stateful rules, the two connection tracking modules outside the classification pipeline are always present, as they have to track all the current connections in order to be ready for state-based rules instantiated at a later time.

TCP state machine. A summary of the TCP state machine implemented in the connection tracking module is shown in Figure 5. 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

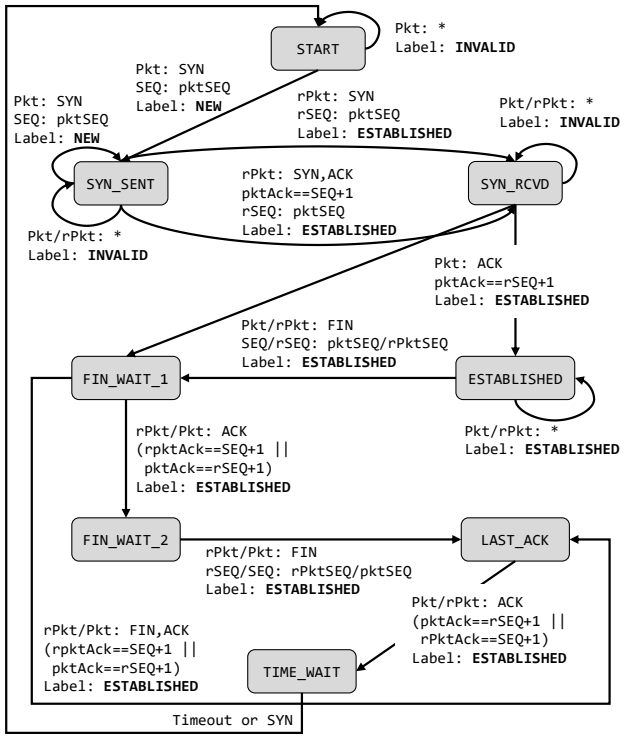


Figure 5: TCP state machine for `bpf-iptables` conntrack. Grey boxes indicate the states saved in the conntrack table; labels represent the value assigned by the first conntrack module before the packet enters the classification pipeline.

connection. Finally, when the connection reaches the `TIME_WAIT` state, only a timeout event or a new `SYN` 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, where a dedicated thread checks the presence of expired sessions. For this reason, the *Conntrack Update* module updates the *timestamp* associated to session each entry when a new packet is received. Since we noticed that the usage of the `bpf_ktime()` helper to retrieve the current timestamp causes a non-negligible performance overhead, we store in a dedicated per-CPU array the current time every *second*, which is used by the data plane to timestamp the session entries. We are confident that a per-second precision is a reasonable trade-off between performance and accuracy for this type of application.

5 CONTROL PLANE

This Section describes the main operations of our control plane, which are triggered whenever one of the following events occur.

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

Algorithm 1 Pre-processing algorithm

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

visible from the root namespace, as shown in Figure 2. This program intercepts all packets flowing through the interface and calls 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, as eBPF programs attached to a NIC cannot be called from other interfaces. 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 the proper set of `netlink` events.

`Netlink` notification. Whenever a new `netlink` notification is received, `bpf-iptables` checks if it relates 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, enabling `bpf-iptables`[5] to inspect its traffic. In the second case, we update the list of local IPs used in the *Chain Selector* with the new address.

Ruleset changes. When the user updates the ruleset, `bpf-iptables` starts the execution of the **pre-processing** algorithm, which calculates the value-bitvector pairs for each field; those values are then inserted in the new eBPF maps and the new programs are created on the parallel chain. The pre-processing algorithm (pseudo-code in Algorithm 1) works as follows. 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 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. 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. 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.

[5]There is a transition window between the reception of the `netlink` notification and the load of the *redirect* program, during which the firewall is not yet active. As far as the eBPF is concerned, this transition cannot be totally removed.

6 EVALUATION

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) [22], 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 testing `bpf-iptables`, we disabled all the kernel modules related to `iptables` and `nftables` (e.g., `x_tables`, `nf_tables`) and the corresponding connection tracking modules (i.e., `nf_conntrack` and `nft_ct`). 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.

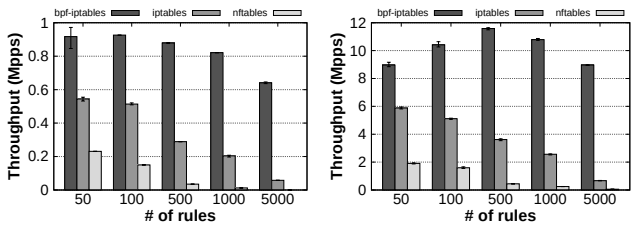


Figure 6: Single 6(a) and multi-core 6(b) comparison when increasing the number of loaded rules. Generated traffic (64B UDP packets) is uniformly distributed among all the rules.

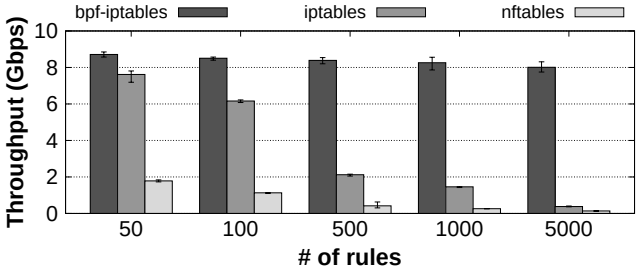


Figure 7: Performance of the `INPUT` chain with an increasing number of rules. `bpf-iptables` runs on a single CPU core and `iperf` on another core.

Testing tools. UDP tests used `Pktgen-DPDK` v3.5.6 [14] and `DPDK` v18.08 to generate traffic, while for TCP tests we used either `iperf` v2.0.10 or `weighttp` [3] v0.4 to generate a high number of *new* parallel TCP connection towards the DUT, counting only the successful completed connections [23]. Particularly, the latter reports the actual capability of the server to perform real work.

Rulesets and Packet-traces. We used the same ruleset for all the firewalls under consideration. In particular, `nftables` rules have been generated using the same rules loaded for `bpf-iptables` and `iptables` but converted using `iptables-translate` [4]. Since synthetic rulesets vary depending on the test under consideration, we describe their content in the corresponding test's section. Regarding the generated traffic, we configured `Pktgen-DPDK` to generate traffic that matches the configured rules; also in this case we discuss the details in each test description.

6.2 System benchmarking

This Section evaluates the performance and efficiency of individual `bpf-iptables` components (e.g., conntrack, matching pipeline).

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. We generated five synthetic rulesets with rules matching the TCP/IP 5-tuple and then analyzed a first scenario in which rules are loaded on the `FORWARD` chain (Section 6.2.2) and a second that involves the `INPUT` chain (Section 6.2.3). In the first case, performance are influenced by both the classification algorithm and the TCP/IP stack bypass; in the second case packets are delivered to a local application, hence the performance are mainly influenced by the classification algorithm.

6.2.2 Performance dependency on the number of rules (`FORWARD` chain). This test loads all the rules in the `FORWARD` chain and the DUT is configured as router in order to forward all traffic received from one interface to the other. The generated traffic is uniformly distributed among all the rules[6], without any packet hitting the default rule. Since each rule is a 5-tuple, the number of TCP generated flows is equal to the number of rules.

Evaluation metrics. We report the UDP throughput (in Mpps) averaged among 10 different runs. This value is taken by adjusting the sending rate not to exceed 1% packet loss. 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 available cores in the DUT are used.

Results. Figure 6(a) and 6(b) show respectively the single-core and multi-core forwarding performance results. We can notice from Figure 6(a) how `bpf-iptables` outperforms `iptables` by a factor of two even with a small number of rules (i.e., 50); this gap is even larger with `nftables`, which is almost 5 times slower in the same conditions. The advantage of `bpf-iptables` is even more evident with more rules; the main performance bottleneck is the scanning of the entire bitvector in order to find the final matching rule, whose size depends on the number of rules (Section 4.4). Finally, Figure 6(b) shows how `bpf-iptables` scale across multiple cores; the maximum throughput is achieved with 1K rules since the number of generated flows with a smaller number of rules is not enough to guarantee uniform processing across multiple cores (due to the RSS/RFS feature of the NIC), with a resulting lower throughput.

6.2.3 Performance dependency on the number of rules (`INPUT` chain). This test loads all the rules in the `INPUT` chain; traffic traverses the firewall and terminates on a local application, hence following the same path through the TCP/IP stack for all the firewalls under testing. As consequence, any performance difference is mainly due to the different classification algorithms. We used `iperf` to generate UDP traffic (using its default packet size for UDP) toward the DUT, where the default `accept` policy causes all packet to be delivered to the local `iperf` server, where we compute the final throughput. To further stress the firewall, we used eight parallel `iperf` clients to generate the traffic, saturating the 40Gbps link.

Evaluation metrics. We report the UDP throughput (in Gbps) among 10 different runs; we forced the firewall to run on a single core, while the `iperf` server runs on a different core.

Results. Figure 7 shows how `bpf-iptables` perform better than the other firewalls, with an increasing gap with larger rulesets. However, the advantage with a low number of rules is smaller compared to the previous case; in fact, in this scenarios, `bpf-iptables` cannot avoid the cost of passing through the TCP/IP stack (and the allocation of the `sk_buff`). Therefore its performance advantage is given only by the different classification algorithm, which is more evident when the number of rules grows. However, it is important to note that in case of `DROP` rules, `bpf-iptables` discards the packets far before they reach the local application, with a sensible performance advantage thanks to the early processing of XDP.

[6]We used a customized version of Pktgen-DPDK [26] to randomly generate packet for a given range of IP addresses and L4 port values.

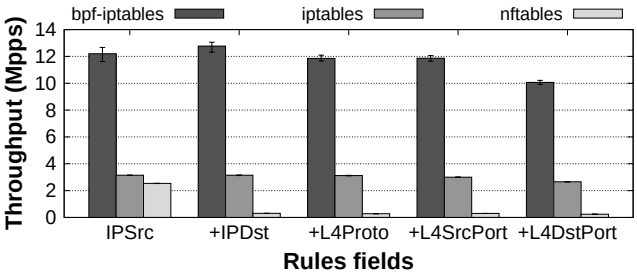
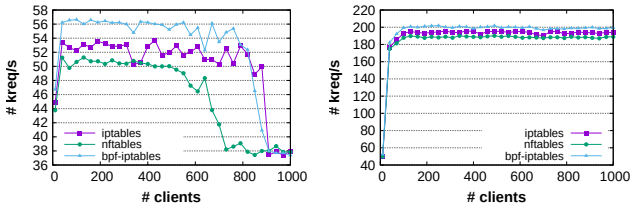


Figure 8: Multi-core performance comparison when varying the number of fields in the rulesets. Generated traffic (64B UDP packets) is uniformly distributed among all the rules.



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

6.2.4 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 increasing 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`.

Test setup and evaluation metrics. Same as Section 6.2.2.

Results. Results in Figure 8 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 depend 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[7].

6.2.5 Connection Tracking Performance. This test evaluates the performance of the connection tracking module, which enables stateful filtering. We used TCP traffic to stress the rather complex state machine of that protocol (Section 4.5) 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 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[8].

Ruleset. This ruleset is made by three rules loaded in the `INPUT` chain, hence operating only on packets directed to a local application. The first rule *accepts* all packets belonging to an `ESTABLISHED` session; the second rule *accepts* all the `NEW` packets coming from the outside and with the TCP destination port equal to 80; the last rule *drops* all the other packets coming from outside.

Evaluation metrics. We measure the number of successfully completed requests; in particular, `weighttp` increments the above number only if a request is completed within 5 seconds.

Results. `bpf-iptables` scores better 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 9(a) and 9(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 Linux kernel.

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

[7] 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.

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

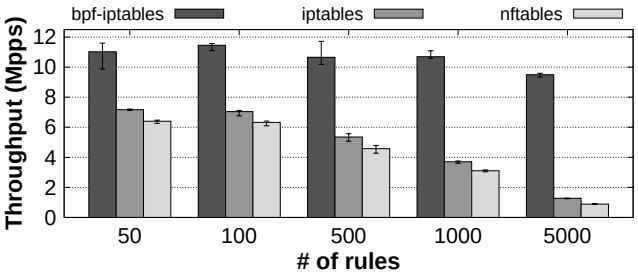


Figure 10: Throughput when protecting a variable number of services within a DMZ. Multi-core tests with UDP 64B packets, bidirectional flows.

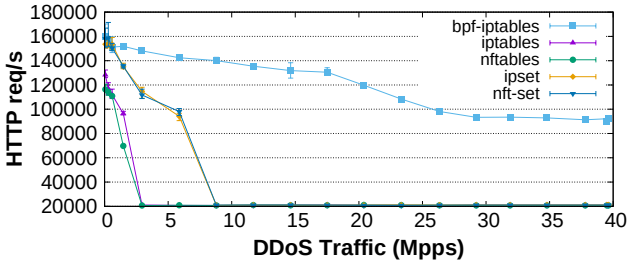


Figure 11: Multi-core performance under DDoS attack. Number of successful HTTP requests/s under different load rates.

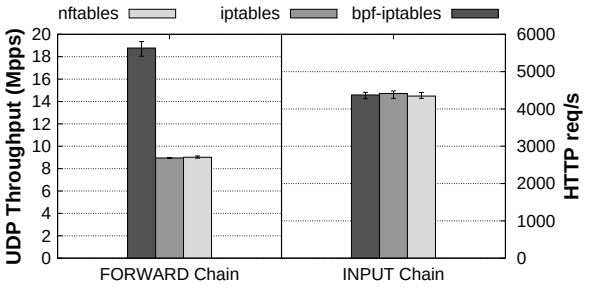


Figure 12: Performance with single default `ACCEPT` rule (baseline). Left: UDP traffic, 64B packets matching the `FORWARD` chain. Right: number of HTTP requests/s (downloading a 1*MB* web page), TCP packets matching the `INPUT` chain.

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 between the two (public and protected) networks. When traffic belonging to a specific flow is seen in both directions, the session is considered `ESTABLISHED` and then will match the first rule of the ruleset.

Evaluation metrics. The test has been repeated 10 times; results report the throughput in Mpps (for 64B UDP packets).

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 10. 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 10 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.

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 blacklisted 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. Figure 11 shows that the performance of `bpf-iptables`, `ipset` and `nft-set` are similar for of low-volume DDoS attacks; `iptables` and `nftables` are slightly worse because of their inferior matching algorithm. However, with higher DDoS load (> 8Mpps), the performance of `ipset` and `nft-set` drop rapidly and the server becomes unresponsive, with almost no requests 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` can successfully sustain ~95.000 HTTP requests/s of legitimate traffic when the DDoS attack rate is more than 40Mpps, i.e., ~60% of the maximum achievable load. Higher DDoS load was not tested because of a limitation of our traffic generator.

6.4 Microbenchmarks

6.4.1 Baseline performance. This test analyzes the overhead of `bpf-iptables` on a vanilla system, without any firewall rule. 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, such as the connection tracking and the logic that applies the default action to all packets (i.e., `ALLOW`). The left side of Figure 12 shows the performance of `bpf-iptables`, `iptables` and

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.

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 12, 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 advantages seen in the previous case, it does not show any worsening either, hence demonstrating that the overhead of the running components is definitely limited.

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 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 compile and inject the new eBPF classification pipeline in the kernel; during this time, bpf-iptables continues to process the traffic according to the old ruleset, with the *swapping* performed only when the new pipeline is ready[9]. Finally, t3 is the time required to delete the old chain, which has no impact on the user experience as the new pipeline is already filtering traffic after t2. Finally, the last column of Table 1 depicts the time required to insert a rule handled by HORUS (Section 4.4.2). Excluding the first entry of this 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.

[9]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.

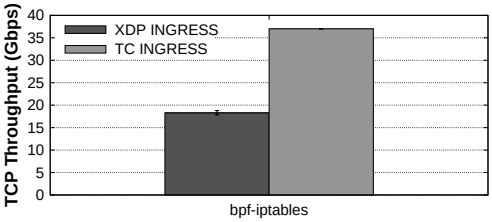


Figure 13: TCP throughput when the bpf-iptables ingress pipeline (with zero rules) is executed on either XDP or TC ingress hook; bpf-iptables running on a single CPU core; iperf running on all the other cores.

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 we can bypass the TCP/IP stack and forward immediately the packet to the final destination (*optimized forwarding*, Section 4.4.2). However, when an eBPF program is attached to the XDP hook, the Generic Receive Offload[10] 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 13, 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 our 40Gbps link[11]. 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 a far more common scenario.

7 ADDITIONAL DISCUSSION

Although one of the main assumption of our work was to rely on a vanilla Linux kernel (Section 2.4), as a possible future work we present here a set of viable kernel modifications that are compatible with bpf-iptables and that may enable new optimizations.

New kernel hooks. Being based on eBPF, bpf-iptables uses a different set of hooks compared to the ones used by the netfilter subsystem (Section 2.1). This introduces the need to predict, in a preceding eBPF hook, some decisions that would be performed only later in the Linux stack. A possible alternative consists in adding new eBPF hooks that operate in netfilter, hence enabling the replacement of selected portions of the above framework that suffer more in terms of performance (e.g., iptables classification pipeline), while reusing existing and well-tested code (e.g., netfilter conntrack). Although this would be the most suitable

[10]Generic Receive Offload (GRO) is a software-based offloading technique that reduces the per-packet processing overhead by reassembling small packets into larger ones.
[11]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.

choice for a 100% iptables-compatible eBPF-based firewall, on the other side it would unavoidably limit the overall performance of the system. In fact, this would set the baseline performance of bpf-iptables to the one of the corresponding TCP/IP stack layer, because of the large amount of code shared between the two approaches and the impossibility to leverage earlier processing provided by the XDP hook. Moreover, the early packet steering provided by XDP enables also the creation of the exact processing pipeline that is required in any given moment in time, instantiating only the proper eBPF modules. This would avoid any source of overhead in the processing path of a packet, which would not be possible in case existing kernel stack components are used.

New eBPF helpers. Adding new eBPF helpers is definitely a suitable direction, in particular with respect to our eBPF conntrack (Section 4.5) that is far from complete and supports only basic scenarios. A dedicated helper would enable a more complete implementation without having to deal with the well-known limitations of eBPF programs (e.g., number of instructions, loops). A similar helper that reused the netfilter connection tracking was proposed in [37], which was at the foundation of an alternative version of bpf-iptables [27]. However, based on the above prototype, we would suggest a custom implementation of the conntrack module in order to be independent from the network stack; the above implementation assumed the use of sk_buff structure and hence was available only to eBPF programs attached to the TC hook.

Improve eBPF internals. One of the biggest limitation of the eBPF subsystem that we faced in this work is the maximum number of allowed instructions, currently constrained to 4K, which limited the maximum number of supported rules to ~8K (Section 4.4, without HORUS). In this respect, the extension of the eBPF verifier to support *bounded loops* would be extremely helpful. However, even though some proposal have been made in this direction (e.g., [15]) the community has to yet found a consensus on how to proceed [11]. A shortcut to this problem is a recent patch [36] that introduced the support for larger eBPF programs up to *one million*[12]; bpf-iptables can benefit from this modification, which would increase the number of supported rules without any change in the overall design of the system.

8 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 filtering 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.

9 ACKNOWLEDGEMENT

We would like to thank the many people who contributed to this work, among the others Pere Monclus, Aasif Shaikh, Massimo Tumolo and the anonymous reviewers for their thoughtful feedback which greatly improved this paper. Our thanks also to VMware and the European Commission (project ASTRID, Grant Agreement no. 786922), 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/ [Online; last-retrieved 29-March-2019].
[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] 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].
[5] Pablo Neira Ayuso. 2018. [PATCH RFC PoC 0/3] nftables meets bpf. https://www.mail-archive.com/netdev@vger.kernel.org/msg217425.html [Online; last-retrieved 29-March-2019].
[6] David Beckett. 2018. Hello XDP_DROP. https://www.netronome.com/blog/hello-xdp_drop/ [Online; last-retrieved 15-November-2018].
[7] D. Borkmann. 2018. net: add bpfilter. https://lwn.net/Articles/747504/ [Online; last-retrieved 30-June-2018].
[8] Jesper Dangaard Brouer. 2018. XDP Drivers. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/drivers.html [Online; last-retrieved 18-September-2018].
[9] Jesper Dangaard Brouer and Toke Høiland-Jørgensen. 2018. XDP: challenges and future work. In *LPC'18 Networking Track*. Linux Plumbers Conference.
[10] J. Corbet. 2009. Nftables: a new packet filtering engine. https://lwn.net/Articles/324989 [Online; last-retrieved 30-June-2018].
[11] Jonathan Corbet. 2018. Bounded loops in BPF programs. https://lwn.net/Articles/773605/ [Online; last-retrieved 29-March-2019].
[12] Jonathan Corbet. 2018. BPF comes to firewalls. https://lwn.net/Articles/747551/ [Online; last-retrieved 29-March-2019].
[13] James Daly and Eric Torng. 2017. TupleMerge: Building Online Packet Classifiers by Omitting Bits. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–10.
[14] DPDK. 2018. Pktgen Traffic Generator Using DPDK. http://dpdk.org/git/apps/pktgen-dpdk
[15] John Fastabend. 2018. Bpf, bounded loop support work in progress. https://lwn.net/Articles/756284/ [Online; last-retrieved 29-March-2019].
[16] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/
[17] 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].

[12]This value indicates the number of instructions processed by the verifier.

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

[19] 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.

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

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

[22] 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].

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

[24] 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.

[25] 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.

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

[27] Sebastiano Miano. 2019. eBPF Iptables with Netfilter conntrack. https://github.com/sebymiano/polycube/tree/iptables_linux_conntrack

[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] Alexei Starovoitov. 2019. bpf: improve verifier scalability. https://patchwork.ozlabs.org/cover/1073775/ [Online; last-retrieved 02-April-2019].

[37] William Tu. 2017. [iovisor-dev] [PATCH RFC] bpf: add connection tracking helper functions. https://lists.linuxfoundation.org/pipermail/iovisor-dev/2017-September/001023.html [Online; last-retrieved 30-March-2019].

[38] 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.

[39] 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].

[40] 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 also shipped inside the Docker image and can be found inside the directory `tests` of the container.
Moreover, all the instructions needed to replicate the results of the paper are available in this repository:

```
$ github.com/netgroup-polito/bpf-iptables-tests
```

Finally, if the users want to try the prototype without the setup needed to replicate the results of the paper, a comprehensive documentation is available at this URL:

```
$ https://github.com/polycube-network/polycube/
  /blob/master/Documentation/components/iptables/
  /pcn-iptables.rst
```

Source code. This software project is available at this URL:

```
$ github.com/polycube-network/polycube
```