mmb: Flexible High-Speed Userspace Middleboxes

Korian Edeline, Justin Iurman, Cyril Soldani, Benoit Donnet
Université de Liège, Montefiore Institute, Belgium

ABSTRACT

Nowadays, Internet actors have to deal with a strong increase in Internet traffic at many levels. One of their main challenge is building high-speed and efficient networking solutions. In such a context, kernel-bypass I/O frameworks have become their preferred answer to the increasing bandwidth demands. Many works have been achieved, so far, all of them claiming to have succeeded in reaching line-rate for traffic forwarding. However, this claim does not hold for more complex packet processing. In addition, all those solutions share common drawbacks on either deployment flexibility or configurability and user-friendliness.

This is exactly what we tackle in this paper by introducing mmb, a VPP middlebox plugin that allows, through an intuitive command-line interface, to easily build stateless and stateful classification and rewriting middleboxes. mmb makes a careful use of instruction caching and memory prefetching, in addition to other techniques used by other high-performance I/O frameworks. We compare mmb performance with other middlebox solutions, such as kernel-bypass framework and kernel-level optimized approach, for enforcing middleboxes policies (firewall, NAT, transport-level engineering). We demonstrate that mmb performs, generally, better than existing solutions, sustaining a line-rate processing while performing large numbers of complex policies.

1 INTRODUCTION

Global Internet traffic has constantly increased over the past decade. In 2017, 17.4 billions of devices have generated more than 45,000 GB/second Internet traffic. By 2022, the number of devices connected to IP networks will reach 28.5 billions, and their traffic will attain 150,700 GB/second, with hours peaking up to a x4.8 increase factor.

In parallel, the traditional TCP/IP architecture (i.e., the end-to-end principle) is becoming outdated in a wide range of network situations. Indeed, corporate networks [20], WiFi hotspots, cellular networks [22], but also Tier-1 ASes [8] are deploying more and more *middleboxes* in addition to traditional network hardware. Indeed, middleboxes can be deployed for, e.g., security (IDS, NATs, firewalls) and network performance (load balancer, WAN optimizer).

Internet actors have thus to deal with this double increase at many levels, and particularly, in building high-speed networking solutions. A wide range of Kernel-bypass I/O frameworks are available to answer this increasing bandwidth demand, and the Linux kernel has been striving to stay afloat [3, 13]. Many of those efforts claim to have succeeded in reaching line-rate for traffic forwarding, less so for more complex packet processing (e.g., a firewall with a large number of rules, TCP options). Moreover, all of them share common drawbacks, on either deployment flexibility by necessitating expensive hardware or specific OS to maintain reasonable performances, or configurability and user-friendliness by requiring non-trivial programming for basic adaptation of common network functions.

In this paper, we overcome those limitations by introducing mmb (**M**odular **M**iddle**B**ox), an open-source [1] plugin for the Vector Packet Processing (VPP [18]) kernel-bypassing framework. mmb aims at achieving line-rate forwarding performance while performing a large number of complex packet manipulation. It leverages VPP employment of classical and recent advances in packet processing techniques, such as computation and I/O batching, Zero-Copy forwarding, low-level parallelism, and caching efficiency.

Moreover, by implementing combinable generic middlebox policies, configurable from an intuitive command-line interface, mmb allows for out-of-the-box middlebox deployment and easy adaptation. On modern hardware, it is able to hold baremetal-like performance while running on a virtual machine, thanks to PCIe passthrough technologies (i.e., SR-IOV, virtio).

We conduct a thorough comparison of trending high-performance packet processing solutions with mmb, for a selection of simple to complex use cases.[2] We find that, with few hardware restrictions and without the need to write a single line of code, mmb is able to sustain packet forwarding at line-rate speed when enforcing a large set of classification

ANRW '19, July 22, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6848-3/19/07…$15.00
https://doi.org/10.1145/3340301.3341121

[1] https://github.com/mami-project/vpp-mb
[2] More experiments are included in the extended version of this paper [10].

and mangling rules, while other solutions either perform worse, require specific hardware or OS, necessitate expert-level configuration, or have inner design limitations that make them inapplicable.

The remainder of this paper is organized as follows: Sec. 2 provides the required background for this paper; Sec. 3 introduces `mmb`; Sec. 4 evaluates its performances; Sec. 5 positions `mmb` with respect to notable high-performance packet processors; finally, Sec. 6 concludes this paper by summarizing its main achievements.

2 VPP BACKGROUND

Vector Packet Processing (VPP) [4] is a Cisco-developed technology providing a high-performance extensible packet-processing stack running in user space. It implements a full network stack and is designed to be customizable. It can run on I/O frameworks such as DPDK, Netmap [19], or OpenDataPlane (ODP) [17].

VPP leverages techniques such as batch-processing, Receive-side Scaling (RSS) queues, Zero-Copy by allowing userspace applications to have Direct Memory Access (DMA) to the memory region used by the NIC, offloading certain packet processing functions to dedicated hardware, and I/O batching to reduce the overhead of NIC-initiated interrupts. While those techniques have been implemented in other kernel-bypassing frameworks (e.g., FastClick [2]) and have been shown to drastically improve performances [1], VPP attempts to surpass it by introducing parallel processing on multiple CPU cores, to maximize hardware instruction pipelining, alongside an optimal use of CPU caches to minimize the memory access bottleneck. To this end, VPP introduces particular coding practices (e.g., memory prefetching, cache-fitting processing nodes, branch prediction) to maximize low-level parallelism and cache locality.

The VPP packet processing path is based on a directed *forwarding graph* architecture. An example of such a graph is shown in Fig. 1 illustrating the sub-graph used by `mmb`. It is made of cache-fitting, modular *nodes* performing a set of functions (e.g., `dpdk-input`, `ip4-lookup`) to packets, in userspace.

Kernel-bypass frameworks usually rely on a classical run-to-completion approach [2, 15], where each single packet is processed by each function separately. VPP chooses to rely on a per-node batch processing, by systematically using pre-allocated packet batches (i.e., *vectors*). When a function is applied to a packets batch, the first packet causes the function to be loaded in the instruction cache. Then, the following packets are guaranteed to hit the cache, amortizing the cost of the initial cache miss over the whole packet vector. Moreover, this approach gives a priori information on the next data sections to be read, allowing for efficient prefetching strategies.

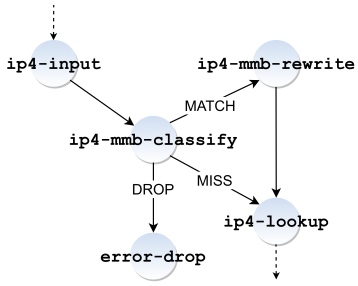


Figure 1: VPP forwarding graph with `mmb` nodes.

Finally, VPP packet processing functions are *N-loops*, which consists in explicitly handling N packets per iteration to increase the code parallelism. This practice aims at exploiting CPU hardware pipelining and at amortizing the cost of instruction cache misses. By unrolling the loop, this algorithm allows for subtracting the processing time of N packets to the fetching time of the N next packets buffers.

3 MODULAR MIDDLEBOX

`mmb` (**M**odular **M**iddle**B**ox) is a VPP extension that performs stateless and stateful classification, and rewriting. It achieves stateless packet matching based on any combination of constraints on network or transport protocol fields, stateful TCP and UDP flow matching, packet mangling, packet dropping and bidirectional mapping. `mmb` is partly protocol-agnostic by allowing to match and rewrite fields `ip4-payload`, `udp-payload`, and `tcp-opt`, and allows for on-the-fly configuration.

3.1 General Overview

Following the VPP architecture, `mmb` forwarding graph consists in two nodes, a *classification* (e.g., `ip4-mmb-classify`) and a *rewrite* node (e.g., `ip4-mmb-rewrite`), as shown in Fig. 1. When `mmb` is enabled, its nodes are simply connected to the processing graph. The classification node is placed right after the `ip4-input` node, that validates the IP4 header checksum, verifies its length and discards packets with expired TTLs.

Depending on the outcome of the classification step, that can either be *drop*, *miss*, or *match*, packets are forwarded respectively to the `error-drop` node which will discard them, to `ip4-lookup`, the node responsible for the Forwarding Information Base (FIB) lookups, that then dispatches packets to the corresponding processing path, or the `mmb-rewrite` node, applying modification rules to packets.

Overall, `mmb` consists in three processing paths that can each be traversed or not by packet vectors, depending on the input policies. A fast path, which relies on VPP bounded index hash tables and implements the mask-based matching operation using binary operators, is shown in Fig. 4. This path is enabled when a rule without any TCP option is entered. Moreover, it restricts the conditions to == (isequal). The stateful flow matching is using this fast path. A first slow path for rules that uses complex conditions (≠, <, >,

```
# mmb <add-keyword> <match>  [<match>  ...  <match>]
                    <target> [<target> ... <target>]

<add-keyword> : add-stateless | add-stateful
<match>       : <field> <condition> <value>
<target>      : mod <field> <value> | add <field> <value>
                | strip [!] <field>  | map <field> <value>
                | shuffle <field>    | drop
```

Figure 2: mmb command-line interface syntax.

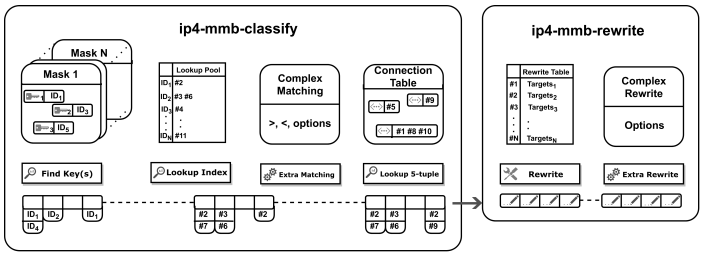


Figure 3: mmb processing path.

≤, ≥), and a second slow path for the linked list parsing required when classifying based on TCP options as well as when rewriting them.

The main goal of mmb is to be easily configurable, and to allow defining a wide range of middlebox policies by combining rules, defined by using commands with a generic semantic [7, 9], at high-speed. To this end, we define a grammar (see Fig. 2) that can be used to build a packet processing middlebox directly from a command-line interface. For example, building a middlebox that rewrites TCP port 80 to port 443 is done as follows:

```
vpp# mmb add tcp−dport 80 mod tcp−dport 443
```

Here is another example of a middlebox stripping all options but MSS and WSCALE if the packet contains the timestamp option:

```
vpp# mmb add tcp−opt−timestamp strip ! tcp−opt−mss strip ! tcp−
     opt−wscale
```

3.2 Classification Node

mmb packet processing is displayed in Fig. 3. The classification node is an extension to VPP classification module, that consists in four distinct steps: a mask-based constraint matching step, an index lookup pool, a complex matching step, and a connection table.

The mask-based matching determines if each packet satisfies constraints on fixed offset fields. For this, we create one classification table per packet mask (e.g., per combination of fields in the match constraint), sized from 16 bytes to at most 80 consecutive bytes. We create one key for a given table per value for its associated packet mask. For each table, the search for a key matching a given packet is a hash-based search performed in constant time.

$$Result_{Classify} = (Packet\ \&\ Mask)\ \oplus\ Key \quad (1)$$

$$Result_{Rewrite} = (Packet\ \&\ Mask)\ |\ Key \quad (2)$$

Figure 4: Binary operations for packet classification and rewrite.

The matching operation consists of two binary operations (AND and XOR), as shown in Fig. 4.1, which are applied to consecutive chunks of 16 bytes, starting from the first non-zero byte in the mask. Each results are OR'ed into a 16-byte variable, that is compared to zero to verify if the matching operation was successful.

Then, for each packet that matched at least one mask-key combination, mmb checks if an additional matching is needed, with a constant-time lookup, and performs it. Additional matching is necessary for constraints on linked-list based fields such as TCP options.

Finally, each packet is matched to a connection table via its 5-tuple. The connection tables keep track of every connection that matched at least one stateful rule, and implements a flag tracking and a timeout mechanism without interruptions. This allows, for example, for reflexive policies. Both TCP and UDP are handled by the connection table.

If a packet matches at least one rule with a drop target, it is immediately forwarded to the `error-drop` node. If the packet matches only non-drop rules, it is forwarded to the `mmb-rewrite` node, and if the packet does not match any rule, it is handed to the next non-mmb node, `ip4-lookup`.

3.3 Rewrite Node

The `mmb-rewrite` node consists in two operations: a mask-based rewrite step that works on the fixed offset fields, similarly to the first step of the classification node, and a complex rewrite step for linked-list based fields.

To perform the rewrite operation, or application of targets, we build a target mask and a target key when the rule is added. The rewrite is then performed with two binary operations (AND and OR), as shown in Fig. 4.2.

mmb allows to perform packet mangling by defining, for any rule, a set of static and dynamic targets. Static targets consists in setting a user-defined value to a chosen field. In the case of TCP options, targets may also define an option strip or an addition. Dynamic targets allows for setting a different value, within a predefined value range or random, on a per-connection basis.

4 PERFORMANCE

4.1 Testbed Description

The testbed consists of three machines with Intel Xeon CPU E5-2620 2.1GHz, 16 Threads, 32GB RAM, Debian 9.0 with 4.9 kernels. Two of these machines play the role of Traffic Generators (TGs), while one is the Device Under Test (DUT). An additional machine with Intel Xeon CPU E5-2630

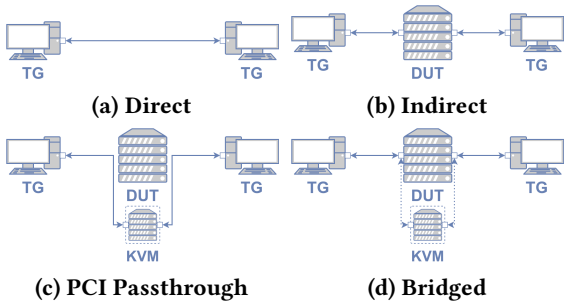


Figure 5: Measurement Setups. TG = Traffic Generator. DUT = Device Under Test. Plain arrows are physical connections, Dotted arrows are bridge networks and the machine surrounded by dots is a virtual machine.

2.4GHz 16 Threads, 16GB RAM, Ubuntu Server 18.04 with 4.15 kernel, is used as alternative DUT for experiments requiring a more recent kernel. Each machine is equipped with an Intel XL710 2x40GB NIC connected to a Huawei CE6800 switch using one port each for TGs and both for the DUT.

The DUT runs VPP 18.10, DPDK 18.08 with 10 1GB huge pages, and a kvm hypervisor with a Ubuntu 18.04 guest. The TGs run iperf3 [21]. The DUTs are configured to maximize their performances.

The testbed has four different setups: A *direct* client-to-server communication setup, shown in Fig. 5a, that is used to evaluate bandwidth baselines and rule out sender-bounded experiments. An *indirect* setup, Fig. 5b, in which the DUT forwards traffic between sender and receiver. A *PCI passthrough* setup, Fig. 5c, allowing the hypervisor to directly connect the NIC to the guest OS. Finally, a *bridged* setup, Fig. 5d, where the guest OS interfaces are connected to the host OS interfaces using two bridges.

We choose to use 7 iperf client-server pairs, in order to analyze the effect of a small amount of large flows, whose processing cannot be distributed on all available DUT CPUs. All experiments last for 20 seconds and omit the first second, to avoid transient effects. Packets are sized according to Ethernet MTU. All NICs distributes packets to the RX rings by hashing both IP addresses and ports. Each experiment result is averaged over ten runs for bandwidth measurements, and a thousand runs for CPU measurements.

4.2 Experiments

The experiments consist in comparing `mmb` to FastClick [2], XDP [13], and iptables. We evaluate two Linux kernel versions (i.e., 4.9 and 4.15) for both `mmb` and iptables because they exhibit significant performance differences. Below, we describe the compared tools.

FastClick [2] is a packet processor framework based on the Click modular router [16]. It comes with multi-queue support, zero-copy forwarding, I/O and computation batching, and integrates both DPDK and Netmap [19]. It also eases

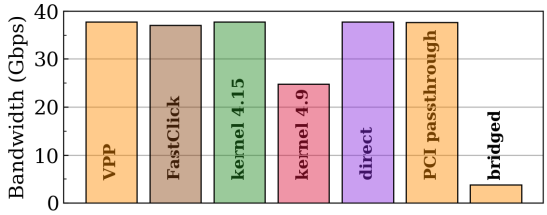


Figure 6: Forwarding baselines. In PCI passthrough and bridged setups, DUT is running VPP.

the writing of Click configurations, as the framework can handle some level of parallelization automatically, without requiring the user to allocate resources manually as in the other Click-based frameworks.

eXpress Data Path (XDP) [13] is a programmable kernel packet processor for Linux. It consists in an extra filtering step in the TCP-IP stack, based on extended Berkeley Packet Filters (eBPF), which are able to perform stateless lookups, flow lookups, and flow state tracking. The main use cases of XDP are pre-stack DDoS filtering, forwarding, load balancing, and flow monitoring. Because eBPFs are introduced in the 4.14 kernel, we only evaluated XDP on the Ubuntu Server 18.04 DUT.

iptables is the builtin Linux firewall. It consists in multiple filtering hooks positioned strategically in the networking stack, that are triggered by packets as they progress in the stack. However, the filtering is performed sequentially and the packets that matches drop rules are not necessarily dropped immediately and might stay longer in the processing pipe. It comes with a connection tracking system, `conntrack`.

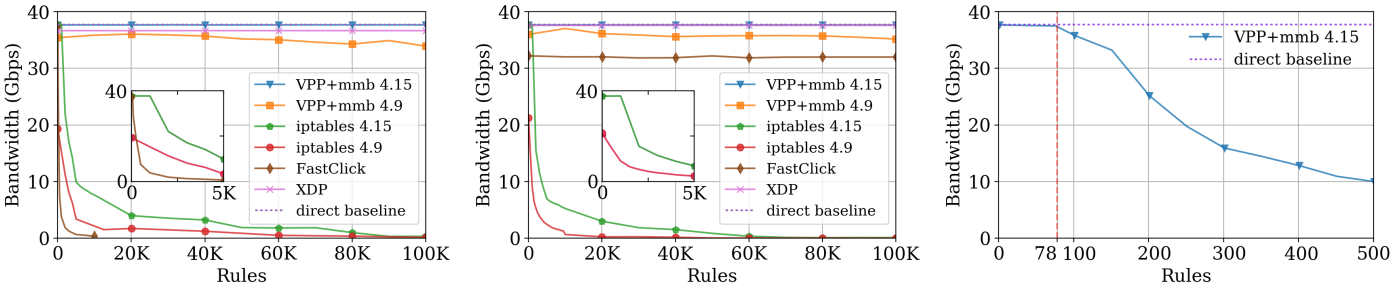
The following use cases are considered[2]: (*i*) packet forwarding, (*ii*) firewall-like packet filtering, (*iii*) packet filtering with stateful flow tracking, and (*iv*) TCP options filtering.

4.3 Results

4.3.1 Forwarding. We first evaluate the TG bottleneck, by running iperf using the *direct* setup. We obtain a 37.7 Gbps throughput baseline. Then, we evaluate VPP, FastClick, and kernel forwarding baselines for the *indirect* setup, and the VPP forwarding baseline for the *PCI passthrough* and *bridged* setups. The results are displayed in Fig. 6.

We observe that VPP, FastClick, and Linux kernel 4.15 forward packets at more than 99% of the direct baseline. The Linux kernel 4.9 performs substantially worse, forwarding only at 24.8 Gbps.

When running VPP, both the *indirect* and *PCI passthrough* setups reach the *direct* baseline. Both setups continue to behave similarly in following experiments. We note that this advocates in favor of `mmb` deployment flexibility and from now on, we report a single result that stands for both setups. Unsurprisingly, the *bridged* performs very poorly at 3.6 Gbps, emphasizing so the importance of direct I/O.



(a) Firewall 5-tuple filtering **(b) Stateful matching** **(c) TCP Options matching**

Figure 7: Performances in *indirect* setup.

4.3.2 Firewall. We configure `mmb` as a firewall and compare it to a FastClick firewall configuration, an XDP-based firewall, and the kernel forwarding with iptables filtering, to evaluate their applicability to a basic firewall-like packet filtering use case. To this end, we generate rules that classify packets based exclusively on five-tuples, to enable tools with mask-based hash classification approaches (e.g., `mmb` and XDP) to use a single table. We generate a new set of rules for every experiment and ensure that no rules are matching the traffic from the TGs. In the real world, these middlebox policies can be used as a firewall as well as DDoS protection measures. We inject these rules to `mmb` as stateless rules.

For the XDP firewall, we use a BPF hash map with five-tuples as keys, and use it to store the type of rule (i.e., accept or drop) and the count of accepted and dropped packets. Every received packet is checked against the BPF for an existing entry. If it exists, the related drop counter is incremented and the packet dropped. Otherwise, the packet passes.

For FastClick, we use an `IPFilter` element that drops packets that match a rule (i.e., none in our test), and will pass them to the routing table otherwise. As `IPFilter` elements can only support up to 2^{16} rules, we have to chain several to support more rules.

With iptables, we inject the rules to the `FORWARD` chain.

The bandwidth results are shown in Fig. 7a. It shows that XDP and VPP with `mmb` on a 4.15 kernel, keep a constant forwarding rate, regardless of the number of rules, both performing very close to the direct baseline. The `mmb` firewall on a 4.9 kernel shows signs of rule count dependent performance, but we believe this is rather due to the kernel.

FastClick performance decreases even more quickly with the number of rules (no data is depicted for more than 10,000 rules because the slow processing stalls the TGs). This is due to the implementation of the `IPFilter` element that, on the contrary to `mmb` and XDP that use a $O(1)$ hash-based approach to match packets to rules, uses a binary search which requires $O(\log_2 n)$ comparisons. Moreover, the matching code has bad cache locality, further contributing to the performance drop.

iptables on a 4.15 kernel surprisingly sustains a line-rate bandwidth until 1,000 rules are inputted, while iptables on 4.9 kernel performance decreases already with very few rules.

This experiment indicates that the `mmb` mask-based *fast path*, when relying on a single table, has a very limited impact on the maximum achievable bandwidth of the forwarding device, regardless on the number of rules.

4.3.3 Stateful. Next, we evaluate the chosen tools against a packet filtering with stateful flow tracking use case. We generate sets of rules matching on the received packets five-tuple, similarly to the previous experiment, and we input a static set of rules to guarantee that all traffic from the TGs is matched, to enable flow tracking capabilities of the tested tools. In the real world, this type of middlebox policies can be used for private network-initiated reflexive ACLs.

We inject these rules to `mmb` as stateful rules in order to have every packet matching at least one rule to add an entry to the connection table. All packets are also checked against the opened connections, whose states are updated accordingly.

With XDP, we build a stateful flow tracker using three different BPF hash maps. One for five-tuple matching rules, filled with the randomly generated rules, one for three-tuple matching rules, with the static rules, and one for connections tracking. The latter has hashes of the five-tuples as keys, and maintains flow information (i.e., timestamps, flag-based TCP state, packet counters).

With FastClick, we use an `IPFilter` element for flow filtering and an `IPRewriter` element for connection tracking. Each packet goes through the latter element when entering and leaving the middlebox, triggering the creation of a new flow entry if it is a flow first packet. If a packet matches an existing flow, it is passed directly to the routing table rather than to the `IPFilter`. `IPRewriter` is not thread-safe and will only recognize a return packet as belonging to a flow if it is processed on the same core that created the flow entry. As Receive-Side-Scaling cannot enforce that, we use one `IPRewriter` per core and keep separate flow state for both directions.

iptables is configured as a stateful firewall by enabling `conntrack`, the connection tracking module, and injecting rules to the `FORWARD` chain.

Results are displayed in Fig. 7b. Again, `mmb` on a 4.15 kernel and XDP have constant line-rate performances. iptables on a 4.15 also shows similar performance than for the stateless firewall experiment, while iptables kernel 4.9 with `conntrack` performs worse.

FastClick performs better than for the stateless case, because the costly filtering step is done only for the first packet of each flow. However, its performance is still significantly lower than that of `mmb` or XDP.

4.3.4 TCP Options. Finally, we evaluate the performance of traffic engineering policies that matches TCP Options. The processing of such policies is more complex because it requires linked list parsing for every packet, and the presence and order of TCP Options in a TCP packet is not known a priori. Rules are generated to match on random value of random TCP Options. We do not mangle TCP options because it would disrupt TCP and affect its performance.

FastClick is not tested against this use case because none of the distributed elements is able to match on variable-offset TCP options. XDP is also not tested against this use case because eBPF stack space is limited to 512 bytes, which is exceeded by the task of implementing complex packet parsing.

The bandwidth measurement results, displayed in Fig. 7c, indicates that the threshold of injected TCP Options-based classification rules to sustain line-rate packet forwarding is 78. This is explained by the CPU-time required for linked-list parsing packets.

4.3.5 Limitations. `mmb` sustains line-rate processing for the selected use case. We evaluated its stability limit when handling rules that match on different combination of fields (i.e., one rule per hash table), and found a clear limit of 26 combinations before after which the performances start to diminish. This is explained by the limited cache size and the complexity of the matching algorithm, that is linear with regards to the tables. We advocate that this limitation is largely sufficient for a realistic usage.

XDP runs almost at line rate for both firewall (Fig. 7a) and stateful (Fig. 7b) use cases, which makes it a good in-kernel alternative to `mmb`. However, BPF limited stack space does not allow for more complex packet manipulation, which restricts XDP applicability.

While FastClick is able to sustain line-rate forwarding, combining elements into more complex packet processing is hardened by thread-safeness and core dependence problems. Moreover, it does not support large numbers of middlebox policies.

5 RELATED WORK

Over the years, numerous works have been proposed for fast and efficient packet processing. One can cite *iptables*, *PF_RING* [5] (a software I/O framework that modifies the socket API to avoid buffer reallocation and bypass unnecessary kernel functionalities to improve the performances of packet capture from those of libpcap), *PacketShader* [12] (a GPU-accelerated software router framework, that perform I/O batching and kernel bypass) and *eXpress Data Path* (XDP) [13], a high-performance programmable kernel packet processor for Linux.

The more specific *mOS* [14] is a networking stack for building stateful middleboxes. Its ambition is to provide a high-performance general-purpose flow management mechanism. It comes with an API to allow for building middleboxes applications requiring flow state tracking such as stateful NATs, or payload reassembly such as NIDS/NIPS and L7 protocol analyzers. mOS is based on mTCP [11], a parallelizable userspace TCP/IP stack.

The *Click* modular router is a flexible router framework [16]. It was not specifically designed for high-speed packet processing as it relies on the Linux kernel via system calls for certain tasks, leading so into an increase in processing time. Further, on the contrary to `mmb`, Click requires the user to write C++ classes to build new functionalities. *RouteBricks* [6] brings hardware multi-queue support to Click, and introduce an architecture for parallel execution of router functionalities as a first step towards fast modular software routers. *DoubleClick* [15] integrates PacketShader I/O batching and computation batching. Moreover, it also takes advantage of the non-uniform memory access (NUMA) CPU architecture. *MiddleClick* [1] further enhances FastClick with flow-processing capabilities. It comes, among others, with an optional middlebox-oriented TCP stack.

This paper has shown that VPP with `mmb` performs better than those state of the art solutions for fast packet middlebox processing.

6 CONCLUSION

This paper proposed `mmb` (Modular Middlebox), a high-performance modular middlebox, implemented as a VPP plugin. `mmb` can be used to deploy out-of-the box middleboxes and to easily and intuitively configure custom policies through its command-line interface, on the contrary to state-of-the-art solutions usually requiring dedicated hardware, specific OS or non-trivial programming.

We compared `mmb` to other high-speed packet processors and demonstrated, through several use cases, that `mmb` is able to sustain packet forwarding at line-rate speed when applying a large number of diverse and complex classification and mangling rules. `mmb` is open source and freely available.[1]

REFERENCES

[1] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy. 2018. Buliding a Chain of High-Speed VNFs in No Time. In *Proc. IEEE International Conference on High Performance Switching and Routing*.

[2] T. Barbette, C. Soldani, and L. Mathy. 2015. Fast Userspace Packet Processing. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[3] J. D. Brouer and T. Høiland-Jørgensen. 2018. XDP – Challenges and Future Work. In *Proc. Linux Plumbers Conference*.

[4] Cisco. 2002. Vector Packet Processing (VPP). (2002). See https://fd.io.

[5] L. Deri. 2004. Improving Passive Packet Capture: Beyond Device Polling. In *Proc. International System Administration and Network Engineering Conference (SANE)*.

[6] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. Symposium on Operating Systems Principles (SIGOPS)*.

[7] K. Edeline and B. Donnet. 2015. Towards a Middlebox Policy Taxonomy: Path Impairments. In *Proc. IEEE International Workshop on Science for Communication Networks (NetSciCom)*.

[8] K. Edeline and B. Donnet. 2017. A First Look at the Prevalence and Persistence of Middleboxes in the Wild. In *Proc. International Teletraffic Congress (ITC)*.

[9] K. Edeline and B. Donnet. 2017. An Observation-Based Middlebox Policy Taxonomy. In *Proc. ACM SIGCOMM CoNEXT Student Workshop*.

[10] K. Edeline, J. Iurman, C. Soldani, and B. Donnet. 2019. *mmb: Flexible High-Speed Userspace Middleboxes.* cs.NI 1904.11277. arXiv.

[11] J. EunYoung, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. 2014. mTCP: a Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[12] S. Han, K. Jang, K.S. Park, and S. Moon. 2010. PacketShader: A GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM*.

[13] T. Høiland-Jørgensen, J. D. Brouer, d. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proc. ACM SIGCOMM CoNEXT*.

[14] M. A. Jamshed, Y. G. Moon, D. Kim, D. Han, and K. S. Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[15] J. Kim, S. Huh, K. Jang, K. S. Park, and S. Moon. 2012. The Power of Batching in the Click Modular Router. In *Proc. Asia-Pacific Workshop on Systems*.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems* 18, 3 (August 2000), 263–297.

[17] Linaro Networking Group (LNG). 2013. OpenDataPlane (ODP). (2013). See https://www.opendataplane.org.

[18] L. Linguaglossa, D. Rossi, S. Pontarelli, D. Barach, D. Marjon, and P. Pfister. 2018. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine* 56, 12 (December 2018), 97–103.

[19] L. Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX Security Symposium*.

[20] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. 2012. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM*.

[21] A. Tirumala, F. Qin, J. Duagn, J. Ferguson, and K. Gibbs. 2005. Iperf, the TCP/UDP Bandwidth Measurement Tool. (2005). See http://iperf.sourceforge.net/.

[22] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. 2011. An Untold Story of Middleboxes in Cellular Networks. In *Proc. ACM SIGCOMM*.