IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 13, 2021, accepted December 27, 2021, date of publication January 5, 2022, date of current version January 13, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3140522

Surgical DDoS Filtering With Fast LPM

DENIS SALOPEK[1], (Student Member, IEEE), MARKO ZEC[1], MILJENKO MIKUC[1], (Member, IEEE), AND VALTER VASIĆ[2]
[1]Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
[2]Republic of Croatia, Information Systems Security Bureau, 10000 Zagreb, Croatia

Corresponding author: Denis Salopek (denis.salopek@fer.hr)

ABSTRACT Can software-based packet filters effectively dampen volumetric distributed denial-of-service (DDoS) streams in an era when 10 Gbps links are considered slow? The potential of longest prefix matching (LPM) for enforcing precise DDoS scrubbing policies seems to be overlooked in contemporary packet filtering datapaths, and in this paper, we argue that this should not be the case by showing that effective whitelist / blacklist LPM-based filtering can be performed with commodity hardware. A showcase datapath we propose can evaluate multiple queries in large separate LPM databases for each forwarded 64-byte packet, while sustaining 10 Gbps line rate on a single CPU core, with a healthy scaling potential due to its lockless architecture and small memory footprint of LPM structures. We demonstrated forwarding 64 million packets per second using only six CPU cores while performing independent lookups for each packet in three large LPM databases created by aggregating malicious IP addresses or by mapping different geolocation identifiers to IPv4 prefixes.

INDEX TERMS Firewalls, network security, packet lookup and classification, software routers.

I. INTRODUCTION

The proliferation of still predominantly IPv4-based volumetric/flooding distributed denial-of-service (DDoS) attacks [1], which are exploiting the openness and simplicity of the Internet's addressing and routing architecture, is placing an increasing burden on Internet service providers (ISP) and datacenter operators. An effective mitigation strategy has to include scrubbing malicious from legitimate traffic close to the attack target. Packet filtering using specialized hardware such as ternary content addressable memories (TCAMs) offers high throughput, but TCAMs have a rigid structure, low density, suffer from high power consumption, and are costly. In a quest for more flexibility and virtualization capabilities, a new interest in implementing high-speed packet filters in software has recently emerged.

Legacy software firewalls available in general-purpose operating systems (OS) were designed when speeds of 100 Mbps and 1 Gbps were considered fast, but in today's datacenters, even 10 Gbps network interface cards (NICs) are gradually being replaced by 25, 40, or 100 Gbps parts. Moreover, software datapaths have evolved towards generalizations such as OpenFlow [2], which aim to adapt to all conceivable packet manipulation scenarios, with emphasis on stateful operation. However, the precise flow tracking / caching paradigm suffers from a wide spectrum of inherent architectural limitations [3]. Particularly in the context of volumetric DDoS attacks, as source addresses of inbound packets are either randomized (spoofed), or the packets originate from vast pools of compromised or vulnerable hosts, the elastically expanding flow tracking structures either quickly reach their preset limits, or spill over CPU's caches. Furthermore, synchronizing access to mutable shared data structures such as flow tables can require tens to hundreds of CPU clock cycles per table access, itself consuming the entire per-packet time budget for a single 10 Gbps link.

The need for simplifying and stripping down packet processing software datapaths of non-essential functions is well recognized and can be reflected in the widespread adoption of fast packet I/O frameworks such as DPDK [4] or Netmap [5], which map NIC buffers directly into user space. More recently proposed mechanisms for fast (pre)processing of packets before they get encapsulated and consumed by complex data structures and function call paths in an OS kernel are currently enjoying gigantic momentum: eBPF/XDP [6], [7], are re-exploring the paradigm of safe just-in-time (JIT) translation of packet filtering programs from bytecode form to native machine code [8] within a running

The associate editor coordinating the review of this manuscript and approving it for publication was Hosam El-Ocla.

Linux kernel, refining it to adapt more efficiently to modern CPU architectures, and making it more versatile.

A surprising common denominator among software datapath proposals from the recent literature (user space and XDP-based alike) is that only a few have explored the advances in longest prefix matching (LPM) reported over the past decade. Most user space proposals use the popular DIR-24-8 [9] scheme available as a library in DPDK, while XDP relies on the even older LC-trie, and rare exceptions such as the Kamuee router [10], which uses PopTrie [11] for LPM, do not focus on packet filtering. Therefore, in this paper, we explore whether there are incentives for moving away from the omnipresent DIR-24-8 to more modern LPM schemes in DDoS filtering scenarios, which call for methods to blacklist 10^5 to 10^6 individual compromised host addresses, such as in the well-exposed Mirai DDoS incidents [12] from 2016. In contrast to current trends in XDP-based end host filtering, we were more interested in providing forwarding capacity to move precision DDoS scrubbing further away from end hosts, i.e., towards upstream providers, with the ultimate goal of better protecting the entire downstream infrastructure, including links at the network edge that are often irreparably saturated by volumetric DDoS attacks. Encouraged by reports of other successful user space network function specializations for speed, such as [3], [13], we designed and implemented our own JIT-compiled user space datapath, which we used as a testbed for tuning and evaluating a handful of recent LPM schemes while having them bombarded with synthetic DDoS-type traffic.

A. OUR CONTRIBUTION

We argue and show that the use of LPM offers significant potential for the development of new DDoS scrubbing strategies. Our research indicates that using large datasets as allowlists and blocklists (usually called whitelists and blacklists) allows for precise, high-speed filtering of IPv4 traffic. To support our case, we refine a modern LPM lookup scheme and integrate it into a packet filtering application that allows multiple independent LPM queries to be performed per packet using randomized keys (IP addresses) while maintaining a 10 Gbps line rate on a single CPU core. We defy the currently prevailing view which favors placing high-speed packet filtering functions as JIT-compiled modules in an OS kernel by demonstrating that efficient user space filtering datapaths can be constructed with fewer constraints. Finally, our results are practical: the prototype described here includes all the components necessary for a real-world application: a comprehensive filter description language matched with a fast parser and compiler front-end, as well as a functional control interface with on-the-fly reconfiguration and statistics gathering capabilities.

The rest of the paper is structured as follows. Section II elaborates our choice and refinements of an LPM scheme. Section III presents our datapath and discusses its design and implementation tradeoffs. Section IV provides a performance evaluation with different synthetic ruleset types. Section V places our work in context with related developments in the field. In Section VI, we outline directions for future work with concluding remarks.

II. LONGEST PREFIX MATCHING

As our main objective is to filter traffic based on queries across multiple large IP address datasets, a suitable scheme should have compactly encoded lookup structures (to promote cache locality and conserve memory traffic), searchable through a simple and not overly branchy process (since CPU's out-of-order execution and branch prediction machinery yield diminishing returns when operating on essentially random data patterns). Ideally, it should be versatile enough to support very specific IP prefixes, as well as broader subnet addressing. The scheme should also perform well both with predominantly random traffic patterns, which are characteristic for cases where attackers are successful at source address spoofing, and with more localized patterns typical of large botnets which transmit with legitimate source addresses. Finally, the scheme should not suffer from exceedingly narrow structural limitations.

PopTrie [11] and SAIL [14] are two LPM schemes from the recent literature which stand out as being capable of delivering more than 200 million lookups per second (Mlps) per CPU core in synthetic tests. Both are multiway tries which begin the lookup process by directly indexing a small table with K most significant bits of the key (IPv4 address), where $K = 18$ with PopTrie, and $K = 16$ with SAIL. If the table entry does not indicate a hit, the search continues with up to two 256-way further levels with SAIL, while PopTrie progresses down the sequence of 64-way nodes with a compact encoding which leverages the *popcnt* x86 machine instruction. Another LPM scheme that consistently comes third in PopTrie vs. SAIL standoffs (the two swap on the throne depending on how the benchmarks were conducted) is DXR [15], which is also the oldest of the three. All three schemes were originally devised to work well with BGP-like IPv4 prefix datasets, i.e., to support around 1 million entries, dominated by prefix lengths less specific or equal to /24.

We selected an updated version of DXR [16] as the main LPM scheme for building into our filtering datapath, while introducing further refinements which we describe in the rest of this section. Our previous familiarity with DXR clearly influenced our choice, but the decisive factor was its blend of high LPM throughput with low memory footprint. In order to perform comparative tests we later added implementations of PopTrie[1] (which worked out of the box), SAIL[2] (which we had to implement ourselves because the reference implementation incorrectly constructed lookup structures for prefix lengths exceeding 16, or simply locked up in an infinite loop), and DIR-24-8 (which was easy to distill from the existing DXR's structures).

[1]https://github.com/pixos/poptrie
[2]https://github.com/mengxiang0811/SAIL

TABLE 1. Lookup structure characterization for D16X2R, D16X4R, PopTrie18, SAIL, and DIR-24-8.

				$D=16, X=2$ (D16X2R)				$D=16, X=4$ (D16X4R)				PopTrie	SAIL	DIR248
Table	N	L	δ	μ	ϕ	ϱ	M	μ	ϕ	ϱ	M	M	M	M
gl2_city	3075441	25.0	115503	0.601	0.652	2150439	8.85	0.764	0.558	2069692	10.13	(NC)	(NC)	(NC)
gl2_asn	430965	22.1	66349	0.844	0.424	349632	1.76	0.927	0.331	309076	2.50	(NC)	(NC)	(NC)
gl2_country	338006	24.1	252	0.932	0.239	117553	0.70	0.970	0.155	102750	1.01	2.23	7.70	35.49
blacklist_1	177935	32.0	1	0.835	0.432	344517	1.75	0.934	0.338	278549	2.41	11.35	19.41	41.15
blacklist_2	11212	32.0	1	0.975	0.200	24985	0.29	0.993	0.106	18514	0.49	1.75	3.91	32.44
blacklist_3	1113393	32.0	1	0.680	0.568	2056778	8.41	0.818	0.475	1967409	9.40	53.62	(NC)	(NC)
whitelist_1	82156	32.0	1	0.878	0.389	162218	1.01	0.957	0.295	109258	1.60	6.16	14.25	34.96
whitelist_2	201477	32.0	1	0.738	0.581	392801	2.08	0.895	0.487	269278	2.98	13.92	25.22	38.38
whitelist_3	264363	32.0	1	0.709	0.603	521079	2.59	0.876	0.509	382509	3.50	17.64	27.55	40.99
private	9	11.1	1	1.000	0.125	2	0.13	1.000	0.031	2	0.13	1.00	0.13	32.00

Number of IPv4 prefixes in a dataset N; Average prefix length L; Number of unique labels (next hops) δ; Direct + eXtension table hit ratio μ; Direct + eXtension table footprint size ratio relative to the original DXR ϕ; Number of 32-bit elements in the range table ϱ; Total memory footprint of the lookup structures in MBytes M. (NC) denotes table construction process not completing due to a structural limitation being exceeded.

DXR's lookup process is trivial: the most significant K bits of the key are used to index a primary table. This may either yield a direct hit (an index in a next hop table), or index a corresponding variable-length address range descriptor block, which is searched in logarithmic time using the remaining less significant $R = 32 - K$ bits of the key. The tradeoff between lookup speed and memory footprint depends on the choice of the direct table size (2^K elements).

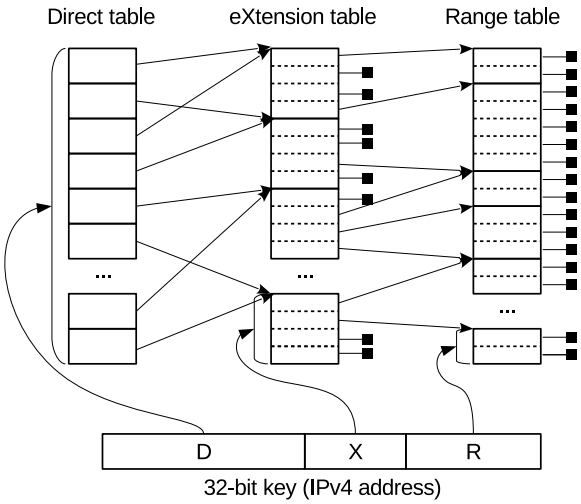

FIGURE 1. Enhanced DXR structures and lookup process.

Inspired by forwarding information base (FIB) compression techniques such as trie folding [17], we further enhanced DXR by splitting the single direct lookup table into two stages, while subjecting the second stage table to data deduplication [18]. As shown in Fig. 1, the lookup process was modified so that the primary (Direct) table is indexed with the D most significant bits of the key, which yields an index pointing to a block in the secondary (eXtension) table, which is then indexed using the subsequent X bits of the key. As in the original scheme, the search either terminates if the indexed eXtension table entry indicates a hit, or it continues with the binary search in a range table chunk using the remaining $R = 32-D-X$ bits of the key.

Table 1 shows lookup structure characterization for several datasets[3] that we experimented with. For the two D, X configurations presented in Table 1, the ratio ϕ of Direct + eXtension table size compared to Direct table size in the original DXR scheme ranges from approximately 0.1 to 0.6, except for the extreme GeoLite2 City table, i. e., for similar memory footprints, the improved scheme permits resolving up to two more bits of the search key via direct indexing, leading to a proportional reduction of (slower) search iterations over range table chunks. The penalty is an extra memory access, but with $D \leq 16$, the indices stored in the Direct table cannot exceed $2^{16} - 1$, so for $D = 16$ the Direct table consumes only 128 KBytes, small enough to achieve good hit rates in L2 caches of modern CPUs. The remainder of the lookup structures should fit into L3 caches, with the parameter X determining the memory footprint / lookup speed tradeoff. The simplicity of the scheme makes inlined compiling of access to the first two tables feasible, which, combined with high hit rates μ reduces the frequency of costly function calls required to complete the resolving process via binary search on address ranges.

Earlier DXR versions limited both the total number of exit labels (next hops) and the length of the range table blocks to 2^{12}, and the overall size of the range table to 2^{19} elements. We revised the encoding scheme to remove the limit on range

[3]Nov 2019 snapshots of free GeoLite2 tables mapping country, BGP AS numbers, and city identifiers to IPv4 prefixes; two smaller blacklists obtained from automated aggregator https://github.com/stamparm/ipsum in Dec 2019 and May 2020 along with a broader one obtained from https://iplists.firehol.org also in May 2020; and three whitelists generated locally from logs of (presumably) legitimate traffic at a department of the University of Zagreb during different periods from Dec 2019 to Mar 2020.

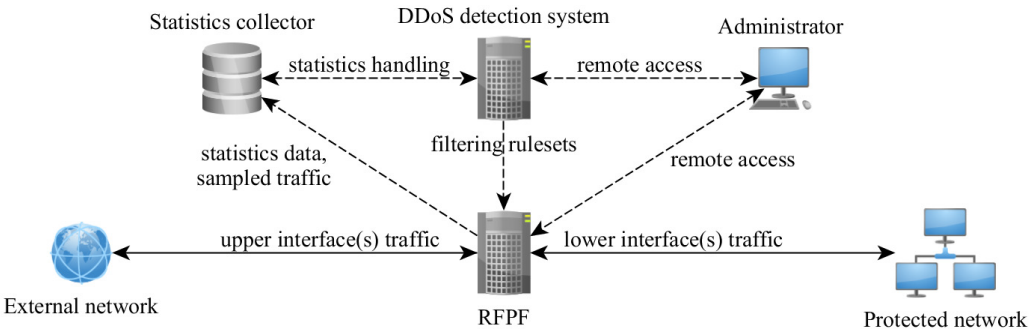

FIGURE 2. A deployment of the filtering system datapath with the working DDoS detection tool.

table block lengths by allowing the values to be stored at the head of the range block in the very rare cases where they would not fit in the eXtension table entries. This also raised the limit on exit labels δ to 2^{D+X}, i.e., to 2^{18} and 2^{20} for the configurations shown in Table 1. The Range table now supports up to 2^{22} elements, which is sufficient for massive datasets such as GeoIP City.

Table 1 further includes memory footprints for PopTrie, SAIL, and DIR-24-8. None of these schemes could accommodate datasets with a large number of exit labels (next hops): PopTrie is limited to 2^{16} and DIR-24-8 to 2^{15} labels, which is a theoretical rather than a practical limitation in many applications, but SAIL's limit of only 2^8 next hops is an even more pronounced one. The capacity for 2^{16} (65,536) and 2^{15} (32,768) next hops is not sufficient for extremely large datasets, but the limit is high enough not to be a concern for reasonably large ones. As can be seen in Table 1, the two largest datasets used for this paper have 115,503 and 66,349 next hops, but they represent extreme cases of dataset fragmentation that can be avoided when using schemes with next hop number constraints. Extending SAIL to reserve more than one byte for encoding next hop information would require major architectural changes and source code upgrades, which would have a negative impact on its performance envelope. Alternatively, to make the comparison with other schemes more fair, next hop encoding space would have to be halved in PopTrie, DIR-24-8 and DXR. Additionally, both DIR-24-8 and SAIL have a fundamental structural problem that limits the number of leaf blocks needed to resolve prefixes more specific than /24 (2^{15} blocks with DIR-24-8 and 2^{16} with SAIL). We had to implement leaf block deduplication to get SAIL and DIR-24-8 to accommodate even our medium-sized datasets, but this enhancement could not save SAIL and DIR-24-8 from both being unable to digest our largest blacklist, which did not present a problem for either PopTrie or DXR.

All the LPM implementations we experimented with were validated by comparing lookup results against the proven reference radix tree [19] implementation borrowed from FreeBSD.

III. FILTERING DATAPATH

A widely accepted approach to "stop" flooding DDoS attacks is BGP blackholing [20]: the victim throws in the towel by telling its upstream provider(s) to stop routing all traffic to the targeted host(s), in an attempt to prevent the rest of its infrastructure from collapsing under the excessive traffic load. Our goal is to enable filtering at such speeds that instead of blackholing the victim's address, providers could precisely filter out vast maps of identified or suspected compromised hosts that serve as the originators of the attack.

Under extreme conditions, such as handling volumetric DDoS traffic, the task of a filtering datapath is to move packets from one interface to another after classifying them and applying appropriate actions as quickly as possible, while still providing elementary operating statistics. Secondary functions may also include diverting manageable amounts of samples to a separate packet processor for detailed analysis. An external tool, such as a DDoS detection system, may use the collected sampled packets and traffic statistics to generate filtering rulesets in the event of an attack, as shown in an example filtering system deployment in Fig. 2. A filtering device does not necessarily have to perform the functions of an IP router; in fact, most legacy firewalls have an option of hiding their presence by not updating the time-to-live (TTL) field of forwarded packets, or they are simply used in transparent bridging mode.

Fig. 3 illustrates the structure of our datapath, which aims to trade richness of functionality for raw speed, dubbed Reduced Feature-Set Packet Filter (RFPF) to emphasize our focus on simplicity and efficiency. We define two sets (bundles) of network interfaces and refer to traffic flowing from the "lower" to the "upper" set as "upstream" and traffic in the opposite direction as "downstream". Traffic in each direction is subjected to entirely independent filtering rulesets. A parser translates the two rulesets into C code as two separate functions that are processed by a standard gcc or clang compiler, yielding executable machine code in a dynamically loadable object. The unit of work for the compiled filtering functions are not individual packets, but

entire packet queues, which netmap exposes to userland applications as circular buffers (or "rings" in netmap / DPDK parlance). This not only allows the compiler to perform extensive optimizations after inlining rule-based classification components within a loop iterating over a receive queue, but also amortizes expensive indirect function calls from worker threads to the dynamically loaded code. Moreover, operating on an entire packet queue leaves the specialized filtering function freedom for optimizations such as elimination of unneeded checks and branches, or choosing a prefetching strategy.

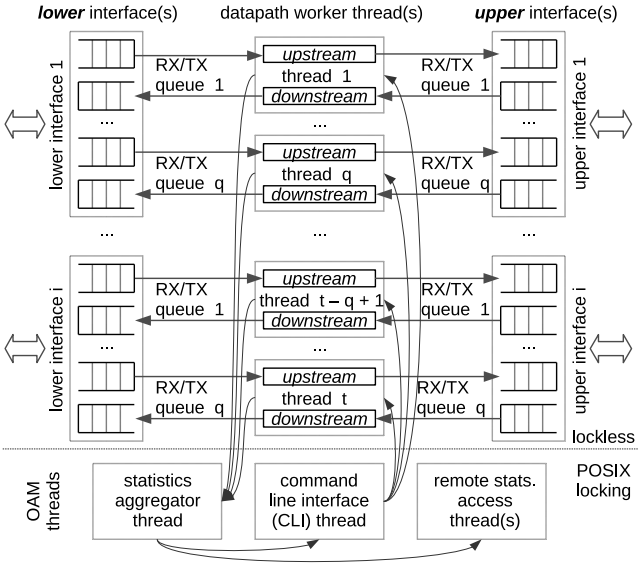

FIGURE 3. RFPF datapath: interfaces, queues, worker threads.

Not unique to our approach but nevertheless an important implicit optimization inherent to compiling rules into machine code is the propagation of constants, such as L4 port numbers, L3 address ranges, etc., into the instruction stream, that reduces data dependencies and CPU pipeline stalls that might otherwise occur when fetching data. Even resolving supposedly minor constants at compile time, such as circular buffer (ring) index masks, can have an observable impact on throughput when total timing budget per packet may not significantly exceed 100 to 150 clock cycles. This is a typical constraint when aiming at 10 Gbps line rate packet forwarding on a single CPU core.

All other components (configuration parser, CLI, etc.) are also implemented in C and operate in user space. Since each NIC can be configured to balance inbound traffic over multiple receive queues (rings), a separate worker thread is allocated to each receive / transmit queue pair between the corresponding "upper" and "lower" interfaces. Optionally, packets can be rerouted or sampled to a "divert" interface bundle.

Two private sets of statistics counters are assigned to each worker thread. At any given time, each worker thread operates on one set, while the other is being accessed by a separate statistics gathering thread, which aggregates per-queue counters into a unified form presentable to the system administrator or external tools. In order to always provide fresh statistics, the aggregator thread switches between the "hot" and shadow counter sets twice per second. Since this is performed at a low frequency, it is feasible to do it by asynchronously updating a shared flag signaling the change to worker threads, with minimal penalty to the filter performance and without losing any data. Worker threads are only permitted read access to shared data, such as LPM tables, which remain immutable for the filter configuration lifetime.

When the packet filter configuration changes, a new set of both shared and thread-local supporting data structures is allocated and populated, and worker threads asynchronously switch over to freshly compiled filtering functions operating on the new state. The transition is non-blocking (i.e. no packets are lost) but is not guaranteed to be atomic, with both old and new state being operated on concurrently as threads switch state, until the last worker thread releases the reference held on the old state in a RCU-like arrangement [21] [22]. Once all threads have completed the transition, the old functions and data structures are relegated to hot-standby state, to be either optionally reactivated on a later request (instant revert functionality), or garbage collected when they are no longer needed. This also permits for several filtering configurations to be prepared upfront and being ready for instant activation should a need arise. Rebuilding filtering routines and all associated data structures from a sample configuration file shown in Listing 1, which includes parsing prefixes from the ASCII format, constructing the LPM structures, compiling the dynamically generated code, and linking it into the running program takes 4.25 seconds on our test machine. Rebuilding a configuration that references blacklist_3 with 1,113,393 addresses instead of the smaller blacklist_1 takes approximately three seconds longer.

The described scheme eliminates the need for any further synchronization between worker threads. The price is that functionalities that inherently require synchronized write access to shared data cannot be easily fit into our datapath without introducing an additional locking mechanism. Expanding RFPF with precise dynamic bidirectional connection tracking or dynamic network address translation (NAT) would require multiple (both upstream and downstream) concurrent lookups and modifications to the flows stored in the shared data. This cannot be done without a locking mechanism, which would inevitably incur a hefty performance penalty.

For the purpose of conducting comparative tests, we have extended our parser frontend to permit optional generation of eBPF/XDP programs based on regular RFPF configurations, in which case the packet datapath is established exclusively within the kernel. However, such generated XDP datapaths still rely on the eBPF's built-in LPM module, which is LC-trie based.

```
include gl2_country.cfg # 338,006 nets, 252 nhops
include whitelist_3.cfg # 264,363 hosts
include blacklist_1.cfg # 177,935 hosts
define UN_SECCNL UK,CN,US,RU,FR

direction downstream
accept src table whitelist_3 any
deny src table black_1 any
accept src table gl2_country HR
sample 0.01 src table gl2_country $UN_SECCNL

direction upstream
deny tcp src port 123-456,789 \
    dst table gl2_country IT,HU
...
```

Listing 1. A sample RFPF configuration file.

IV. PERFORMANCE EVALUATION

A series of tests was conducted in a simple 8 * 10 Gbps Ethernet testbed[4] to characterize the performance envelope of our datapath under different filtering rulesets and operating conditions. Our tests were focused on how throughput scales when the processing load is distributed among multiple processing cores, which we tuned during the experiments by adjusting the number of hardware queues to which the interface cards distribute incoming packets and by gradually turning on 10 Gbps packet sources.

Fig. 4 shows baseline throughputs with unconditional forwarding and dropping rulesets. The software accesses the header of each packet to identify traffic by EtherType ID and count IPv4 and all other protocols separately. The user space datapath can filter out (drop) 29.8 Mpps (millions of packets per second) and forward 28 Mpps using a single CPU core, while the peak throughput, limited by the PCIe 3.0 x8 bandwidth [23] of our NIC's, saturates with four cores when forwarding packets at 74 Mpps and five cores when dropping them at 84 Mpps. The throughput with XDP is considerably lower: 11.5 Mpps when dropping and 7.3 Mpps when forwarding with a single core, which is consistent with other previous reports using hardware similar to ours [6], [24], [25]. XDP dropping throughput scaled up to 48.6 Mpps with eight cores, while forwarding saturates at only 19.1 Mpps with five cores, after which it degrades. The main reason for XDP's lower forwarding throughput is architectural: to forward a packet using XDP, one eBPF program must be executed on the inbound interface, which then places a packet into a software queue, from which another eBPF program attached to the outbound interface dequeues the packet and forwards it to the NIC's hardware queue. In contrast, user space datapath processes packets to completion, i.e., it dequeues packets from an RX queue and moves them in a single loop directly to the TX queue of an outbound NIC. Another factor contributing to XDP's lower throughput figures are the counters, which with eBPF have to be updated via calls to helper functions that check the indices of the counter arrays for their limits. This (an extra function call and a conditional branch) further eats into the miniscule timing budget available for each packet. With the user space datapath, each worker thread can simply update its private set of counters without constraints. Finally, the atomic unit of work for eBPF programs is individual packets, while packets in the RFPF's user space datapath are delivered to filtering functions where they are processed in batches, which also contributes to its more efficient operation.

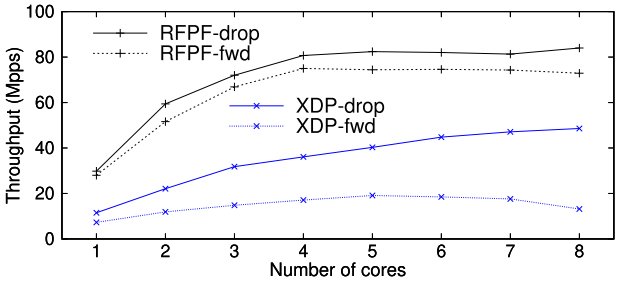

FIGURE 4. Baseline forwarding / dropping throughput.

In subsequent tests, we used only the user space (RFPF) datapath to examine the performance envelopes of the LPM schemes. We used a ruleset configuration derived from the example in Listing 1, which ensured that the source address of each packet was subjected to three LPM queries before the packet was either dropped or forwarded. One test was performed with fully random traffic, the other with a mixture of 10% random traffic, 20% addresses from the whitelist, and 70% addresses from the blacklist. Using this combination of traffic exerts the filter significantly more than random traffic. It is used to show the performance of the filter in the event of a DDoS attack, as this synthetic traffic can be a reasonable substitute for real DDoS traffic.

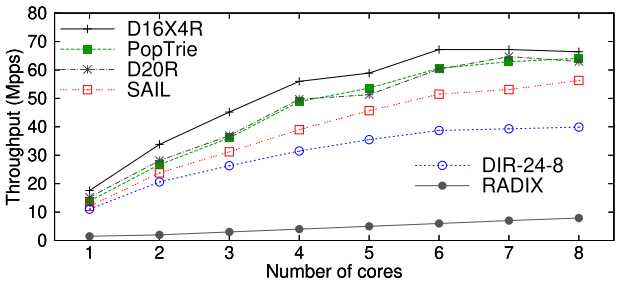

FIGURE 5. Forwarding throughput, random traffic, small blacklist.

Fig. 5 shows the forwarding throughput for uniformly random traffic as a function of active processing cores. The traffic pattern is representative of flooding attacks with successful source address spoofing. The radix tree

[4]Device under test (DUT): 3.6 GHz AMD Ryzen 7 3700X CPU, X470 type motherboard, 8 GB of DDR4 2400MHz RAM split over two memory channels. The CPU has eight physical cores organized in two "core complex" blocks, each having 16 MB of private L3 cache. Simultaneous multithreading (SMT) mode was disabled. The DUT had two quad-port Intel X710-SR2 10 Gbps Ethernet NICs, each plugged in a x8 PCIe 3.0 slot routed directly to the CPU. Four auxiliary machines served as packet sources and / or sinks, each with a single dual-port Intel X520-SR2 10 Gbps NIC. The DUT ran Ubuntu 18.04 with Linux 5.5.0-rc3+ kernel. Aux. machines ran FreeBSD 11.4-RC1.

implementation borrowed from FreeBSD, which has a complexity of $O(W)$, where W is the width of the key (32 bits), is shown for reference and achieves 7.9 Mpps. At 39.9 Mpps, DIR-24-8 was the worst performer of more modern schemes due to its large memory footprint, followed by SAIL, which suffers from a similar property and reached 56.3 Mpps with 8 cores. PopTrie and the original version of DXR (D20R) achieved almost identical maximum throughput (64.1 and 64.7 Mpps with), while peak throughput with the revised DXR (D16X4R) was already saturated with six cores at 67.2 Mpps. The more compactly encoded D16X2R was between PopTrie / D20R and D16X4R (not shown in the chart so as not to overload it).

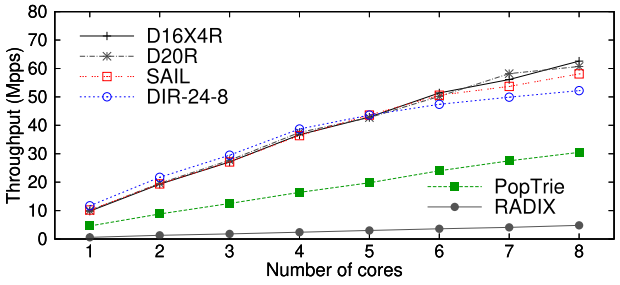

FIGURE 6. Forwarding throughput, botnet traffic, small blacklist.

Fig. 6 shows the results with the same filtering configuration but subjected to synthetic traffic with source addresses being 10% random, 20% from the whitelist, and 70% addresses from the blacklist. The goal of this test was to simulate traffic originating from a large botnet which transmits with legitimate source addresses. While DXR scaled best, reaching 62.6 Mpps, the difference between the LPM schemes was minimal, with the exception of PopTrie, which suffered a substantial hit and achieved only 30.5 Mpps.

We repeated the same sequence of tests but with the (larger) blacklist_3 which encompasses 1.1 million host addresses, versus 178 thousands addresses of blacklist_1 used in the previous tests. As mentioned earlier, neither SAIL nor DIR-24-8 could accommodate such a table due to their structural limitations, so benchmarks were limited to DXR variants, PopTrie, and BSD radix tree. For test traffic with fully randomized source addresses, D16X4R has a healthy lead of 66 Mpps over PopTrie and D20R, which reached 58 and 59.5 Mpps, respectively, as shown in Fig. 7.

However, with the same mix of localized and random traffic as previously used to simulate botnet traffic with legitimate source addresses, throughput degrades for all schemes, as shown in Fig. 8. For DXR, the performance hit is modest (from 66 to 51.2 Mpps with D16X4R and from 59.5 to 48.1 Mpps with the older D20R), but PopTrie suffers from a throughput degradation of approximately 65% to only 21.4 Mpps with eight cores. PopTrie's poor performance can be related to the explosive growth in the size of its lookup structure, which contains numerous host addresses sparsely placed all over the IP address space (see Table 1), which at 53.62 MB are over five times the footprint of D16X4R, and significantly exceed the size of the L3 caches (16 MB).

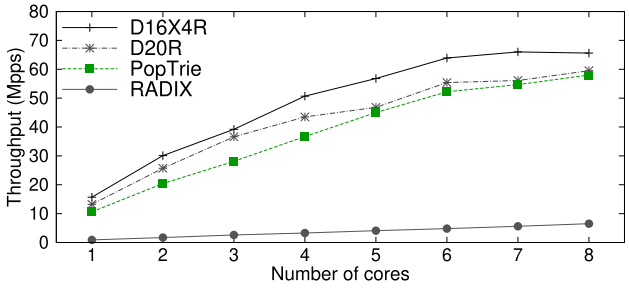

FIGURE 7. Forwarding throughput, random traffic, big blacklist.

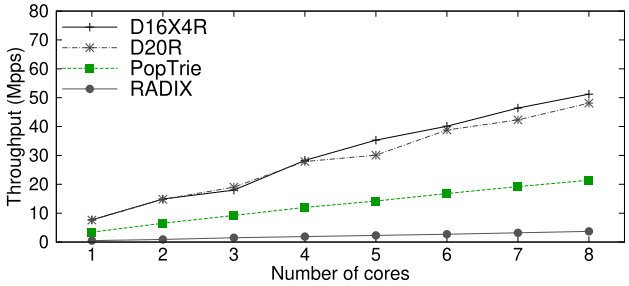

FIGURE 8. Forwarding throughput, botnet traffic, big blacklist.

V. RELATED WORK

The concept of dynamically recompilable datapaths has been actively explored for over two decades: a report on a JIT-compiled BPF optimizer [8] dates back to 1999. More recent refinements include extended BPF (eBPF) and eXpress Data Path (XDP), which were devised with a focus on fast packet processing while enforcing safe execution of potentially untrusted code in the Linux kernel, thus imposing numerous constraints on how the code may be structured in order for the kernel-level verifier to deem it acceptable. By design, eBPF sandboxes accessing any data structures at runtime, so even dereferencing a simple linear table element has to be resolved by calling a function that performs bounds checking on caller-provided indices before returning a pointer to the requested element, thus generating an extra data flow dependency per memory access. Reference [7] reported 24 Mpps baseline packet dropping throughput on a single core, more than double what we achieved with XDP (Fig. 4), using a program that blindly dropped all packets without ever accessing their headers and without updating any counters. The same paper reported a single-core forwarding throughput of about 8.5 Mpps with XDP, while we observed 7.3 Mpps, which is closer to other recent XDP reports such as [6], which range from about 2.2 to 6.6 Mpps.

Mostly focused on end host protection, handling DDoS is the main motivation for many XDP-based proposals. Early reports such as [26] and [27] lack performance evaluation. Reference [24] explores the tradeoff between hardware and

software for end host protection. Their hybrid solution, hardware-assisted preprocessing on a SmartNIC with XDP running in software, allows traffic to be dropped at rates of about 14 to 35 Mpps for 1000 malicious IP addresses using 4 CPU cores. Similarly, [25] advocate eBPF/XDP for end host DDoS protection in combination with a SmartNIC that offloads some processing, reporting packet dropping rates of up to 14.88 Mpps, but lack a performance evaluation for a middlebox-type firewall use case. To take advantage of the NIC offloading, the workload has to be carefully partitioned between the (constrained) NIC hardware and the (more flexible) eBPF software, also noticed by [28], which may indeed work well for simple rulesets with smaller sets of target IP addresses / nets.

A common denominator among most of the packet filtering proposals encountered (XDP-based and others) was that they tended to overlook the potential of leveraging large LPM datasets for devising new DDoS scrubbing strategies. Most user space datapaths that require LPM (e.g., [3]) resort to the DPDK's built-in DIR-24-8 implementation. Reference [7] exercised a single LPM database with 752,138 distinct routes with only 4,000 random IPv4 addresses and observed a single-core throughput of 3.4 Mpps with XDP, which is barely sufficient for forwarding regular traffic at 10 Gbps and significantly less than RFPF with multiple LPM datasets and streams of fully randomized IPv4 addresses. As a notable exception, [29] proposes an approach to DDoS dampening similar to ours: a user space datapath that relies on DPDK for packet I/O and a large LPM dataset for DIR-24-8 [9] based blacklisting. Using a proper testing methodology with randomized traffic and with a single LPM dataset instance, they report forwarding throughput of about 7 Mpps on one CPU core, which is slightly lower compared to our results when evaluating RFPF with DIR-24-8, but again inferior to DXR-based LPM. However, their report does not include any general-purpose firewalling features, nor does it discuss the multi-LPM dataset or multi-core scaling architecture / strategy.

Reference [30] is a kernel-level routing datapath, reaching forwarding throughput of 28.2 Gbps with 64-byte packets (cca. 42 Mpps) on an octa-core Xeon E5 machine and a forwarding table with approximately 280k IPv4 prefixes. However, their performance evaluation only shows results for IPv4 forwarding without any firewall rules.

A more generalized development for defining packet datapaths, compilable for both hardware and software, is P4 [31], a datapath description language. It can be used to construct software firewalls such as [32], which still fall short of demonstrating throughput levels required for facing real-world DDoS scenarios.

VI. CONCLUSION

The spectrum of contemporary DDoS firefighting practices spans from declaring defeat and blackholing victims' addresses via BGP in order to reduce disruptions to other parts of the datacenter infrastructure, to filtering in end hosts before packets enter the network stack, which is where much of the current XDP-based development is taking place. With this paper we wanted to bring the center of this spectrum back into focus, i.e., scrubbing malicious traffic floods using middleboxes, before packets hit end hosts.

We introduced a filtering datapath specialized for forwarding speed and fast LPM: on a consumer-grade 8-core machine, we have demonstrated forwarding traffic at rates exceeding 60 Mpps (i.e., 40 Gbps line rate with 64-byte packets) while subjecting all packets to a series of LPM queries in databases, each encompassing several hundred thousand network prefixes or host addresses. Our experimental evaluation has shown that the choice of LPM scheme makes or breaks the performance of such a filtering datapath, and that some popular LPM schemes may be ill-suited for blacklisting applications with large address datasets due to their inherent structural limitations (insufficient memory for next hop labeling or for more specific prefixes, resulting in inability to load larger datasets). We have shown that even ostensibly minor and simple tweaks to LPM data structures and lookup algorithms may yield real-world throughput gains approaching 10 Mpps, i.e., nearly 20% of the total forwarding capacity of our test system.

REFERENCES

[1] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 4, pp. 2046–2069, 4th Quart., 2013.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[3] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane specialization for high-performance OpenFlow software switching," in *Proc. ACM SIGCOMM*, Aug. 2016, pp. 539–552.

[4] *Intel Data Plane Development Kit (Intel DPDK)*. Accessed: Dec. 29, 2021. [Online]. Available: http://dpdk.org/doc/guides/prog_guide

[5] L. Rizzo, "Revisiting network I/O APIs: The netmap framework," *Commun. ACM*, vol. 55, no. 3, pp. 45–51, 2012.

[6] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with eBPF: Experience and lessons learned," in *Proc. IEEE Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2018, pp. 1–8.

[7] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," in *Proc. ACM Int. Conf. Emerg. Netw. Exp. Tech. (CoNEXT)*, 2018, pp. 54–66.

[8] A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *Proc. ACM SIGCOMM*, 1999, pp. 123–134.

[9] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM*, Mar./Apr. 1998, pp. 1240–1247.

[10] Y. Ohara and Y. Yamagishi, "Kamuee Zero: The design and implementation of route table for high-performance software router," in *Proc. Internet Conf.*, 2016, pp. 1–10.

[11] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 57–70, Aug. 2015.

[12] M. Antonakakis *et al.*, "Understanding the Mirai botnet," in *Proc. 26th USENIX Sec. Symp.*, 2017, pp. 1093–1110.

[13] I. Marinos, R. N. M. Watson, and M. Handley, "Network stack specialization for performance," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 175–186, Feb. 2015.

[14] T. Yang, G. Xie, A. X. Liu, Q. Fu, Y. Li, X. Li, and L. Mathy, "Constant IP lookup with FIB explosion," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1821–1836, Aug. 2018.
[15] M. Zec, L. Rizzo, and M. Mikuc, "DXR: Towards a billion routing lookups per second in software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 29–36, Sep. 2012.
[16] M. Zec and M. Mikuc, "Pushing the envelope: Beyond two billion IP routing lookups per second on commodity CPUs," in *Proc. 25th Int. Conf. Softw., Telecommun. Comput. Netw. (SoftCOM)*, Sep. 2017, pp. 1–6.
[17] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 149–162, Feb. 2016.
[18] M. Zec, "Improving performance in software internet routers through compact lookup structures and efficient datapaths," Ph.D. dissertation, Fac. Electr. Eng. Comput., Univ. Zagreb, Zagreb, Croatia, 2019.
[19] K. Sklower, "A tree-based packet routing table for Berkeley UNIX," in *Proc. USENIX Winter Conf.*, 1991, pp. 93–104.
[20] V. Giotsas, G. Smaragdakis, C. Dietzel, P. Richter, A. Feldmann, and A. Berger, "Inferring BGP blackholing activity in the internet," in *Proc. Internet Meas. Conf. (IMC)*, Nov. 2017, pp. 1–14.
[21] H. T. Kung and P. L. Lehman, "Concurrent manipulation of binary search trees," *ACM Trans. Database Syst.*, vol. 5, no. 3, pp. 354–382, Sep. 1980.
[22] P. E. McKenney, "Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels," Ph.D. dissertation, OGI School Sci. Eng., Oregon Health Sci. Univ., Portland, OR, USA, 2004.
[23] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 327–341.
[24] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese, "Introducing SmartNICs in server-based data plane processing: The DDoS mitigation use case," *IEEE Access*, vol. 7, pp. 107161–107170, 2019.
[25] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle, "Demystifying the performance of XDP BPF," in *Proc. IEEE Conf. Netw. Softw. (NetSoft)*, Jun. 2019, pp. 208–212.
[26] G. Bertin, "XDP in practice: Integrating XDP into our DDoS mitigation pipeline," in *Proc. Tech. Conf. Linux Netw., Netdev*, vol. 2, 2017, pp. 1–5.
[27] A. Deepak, R. Huang, and P. Mehra, "eBPF/XDP based firewall and packet filtering," in *Proc. Linux Plumbers Conf.*, 2018, pp. 1–5.
[28] J. Kicinski and N. Viljoen, "eBPF hardware offload to SmartNICs: cls_bpf and XDP," in *Proc. Tech. Conf. Linux Netw., Netdev*, vol. 1, 2016, pp. 1–6.
[29] E. Kirdan, D. Raumer, P. Emmerich, and G. Carle, "Building a traffic policer for DDoS mitigation on top of commodity hardware," in *Proc. Int. Symp. Netw., Comput. Commun. (ISNCC)*, Jun. 2018, pp. 1–5.
[30] C.-H. Hong, K. Lee, J. Hwang, H. Park, and C. Yoo, "Kafe: Can OS kernels forward packets fast enough for software routers?" *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2734–2747, Dec. 2018.
[31] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.
[32] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4Guard: Designing P4 based firewall," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Oct. 2018, pp. 1–6.

DENIS SALOPEK (Student Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science from the Faculty of Electrical Computing and Engineering, University of Zagreb, Zagreb, Croatia, in 2011 and 2013, respectively, where he is currently pursuing the Ph.D. degree in electrical engineering and computing.

From 2013 to 2016, he worked as a Research Assistant on a project called E-IMUNES, in collaboration with Ericsson Nikola Tesla and since then he has been working as a Teaching Assistant at the Faculty of Electrical Computing and Engineering. He also participates in the project "Smart human-centric services in interoperable and decentralised IoT environments" (IoT4us). His research interests include high-speed networking, FPGA, virtualization, and kernel programming. He is currently the Lead Developer of IMUNES, and has contributed to the Linux and FreeBSD kernels.

MARKO ZEC received the B.Sc. degree in electrical engineering and the Ph.D. degree in computer science from the University of Zagreb, in 1997 and 2019, respectively.

After working as a Systems and Network Administrator, a Designer, and a Consultant at IBM, AT&T, and several local system integration companies. In 2005, he joined the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, where he currently holds the position of an Associate Researcher. His research interests include computer networks, operating systems, and programmable logic. His pioneering work from 2002 at virtualizing networking state in a general-purpose operating system was later merged into the mainline FreeBSD kernel, in 2008, while the concept was later independently embraced by Linux and Solaris as well. At that time novel, the network stack virtualization technology became the foundation for a popular network emulation tool called IMUNES, which he developed together with Prof. Mikuc.

MILJENKO MIKUC (Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Zagreb, Croatia, in 1997.

He is currently an Associate Professor at the Department of Telecommunications, Faculty of Electrical Engineering and Computing, University of Zagreb. His research interests include digital logic design, network protocols, network simulation, and security.

VALTER VASIĆ received the M.Sc. degree in information and communication technology from the University of Zagreb, in 2010, and the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2016. His Ph.D. thesis was "Secure layer-agnostic agreement protocol for cryptographically agile communication."

He was an Associate Researcher at the Faculty of Electrical Engineering and Computing, University of Zagreb, where he has published more than ten articles in journals and conference proceedings. He continued his career as a Software Developer at Ericsson and is now working as a Security Researcher in a computer security incident response and prevention team at Information Systems Security Bureau. His research interests include network communication, information security, network traffic analysis, and virtualization.